How to remove dependencies from large software projects with confidence

Master's thesis

Ching-Chi Chuang

How to remove dependencies from large software projects with confidence

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Ching-Chi Chuang
born in Taipei, Taiwan

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

ING
Bijlmerdreef 24, 1102 CT
Amsterdam, the Netherlands
www.ing.com

How to remove dependencies from large software projects with confidence

Author: Ching-Chi Chuang
Student id: 5304989
Email: C.Chuang-1@student.tudelft.nl

Abstract

Dependency management is an important task in software maintenance. However, identifying and removing unused dependencies takes a lot of effort from developers as existing tools may discover many false positives which are challenging to distinguish. This paper proposes a decision framework to improve unused dependency detection. It is applied to an industrial Maven project. Firstly, OPAL(a call graph tool) augments the call graph of a dependency analysis tool DepClean to support dynamic features of Java. Secondly, the classification of the relationship between dependencies simplifies the comprehension of an unused dependency. Thirdly, a decision process prioritizes the test of removing unnecessary dependencies. Results show that developers can focus their efforts on maintaining bloated dependencies by following the recommendation of the proposed decision process. It is particularly noteworthy that this decision framework helps reduce one-third of false positives of unused dependencies in a given industrial Maven project. In addition, our suggestions are compared to the motive of removing dependencies in three open-source Maven projects. Results indicate that our advice is consistent in the reasoning behind removing dependencies. Hence, this work reduces the effort for developers to decide on dependency elimination.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. L. Cruz, Faculty EEMCS, TU Delft
Company supervisor:	Robbert van Dalen, Vladimir Mikovski, ING Bank
Committee Member:	Dr. C. Lofi Member, Faculty EEMCS, TU Delft

Preface

This work is a memorable journey for me to learn not only how to complete a research study but also how to become an independent researcher like all of my supervisors.
I would like to thank professor Luis. Thank you for spending hundreds of hours helping me find where the contritions of this thesis are and teaching me to write a crystally clear scientific essay. I am also amazed at how you digest information so quickly and precisely.
I would like to thank Robbert. Thank you for brainstorming with me every week in the past year and keeping me on the right track for implementation, presentation, and paper writing. I also thank you for sharing many inspiring concepts and ideas now and then.
I would like to thank Vladimir. Thank you for giving me a chance for the internship and spending countless hours studying this research with me. I also thank you for your warm care both at work and in private.
I would like to thank Maarten. Thank you for consulting me every day in the standup meeting with all your professional suggestions and wisdom. I also thank you for taking care of me well as your team member.
I would like to thank Kevin and Tim. Thank you for always giving me timely help and encouragement at every stage of my internship. I also thank you for treating me not only as a student but also as a friend. It is my honor to be able to participate in both of your PhD defenses during this wonderful year.
I would like to thank Nenad, Megha, and Sonal. Thank you for your support in your extra working hours. Without your kindly contribution, I can not evaluate the performance of this work.
I would like to thank Mauricio and Luna. Thank you for providing this research topic to me and giving me a chance to do my thesis at ING. I am fortunate to be your students.
I would like to thank professor Arie and Christoph for your time and effort in my thesis committee. I also want to thank professor Arie for your suggestion to submit this work to SCAM 2022.
I am so blessed that I can work with all of you in the past year.

Ching-Chi Chuang
Delft, the Netherlands
August 31, 2022

Contents

List of Figures

Chapter 1

Introduction

Modern software systems rely on package managers to gain benefits from the increasing number and massive support of dependencies [12]. Software dependencies hosted on centralized code repositories by package managers allow software engineers to reuse code, reduce development costs and ease maintenance efforts. While the convenience of adding new collections of software dependencies speeds up software development, software projects might retain dependencies that gradually become obsolete throughout the process of development [20]. ependencies that become obsolete and overlooked can increase complexity, decrease maintainability, and in some cases bloat software size. Thus, it is important for developers to properly clean outdated dependencies.

This is a problem we take seriously at ING Bank[1]. ING is a global bank and large software organization that offers financial products and services to 38.5 million customers in over 40 countries [3] and has 15,000 employees in software technology. Hence, at ING it is quintessential that software projects are continuously maintained and meet high-quality standards. Leaving unused dependencies in large software projects can lead to major problems downstream (e.g., in security, maintainability, scalability, etc.). However, deciding to remove a dependency can be an intimidating task: one wrong decision could make core business services temporarily unavailable.

The mainstream approaches to detecting unused dependencies rely on static dependency analysis or dynamic dependency analysis [11]. The performance of the static approaches depends on the soundness and precision of the call graph construction whereas the performance of the dynamic approaches resorts to the coverage of route collections at run time. Hence, static approaches tend to be quicker and more scalable. However, generating a call graph has been considered an undecidable problem [10], meaning that it is difficult to confidently say whether a dependency is being reached or not based on state-of-the-art call graph generation tools.

Several static analysis tools have been developed to remove unused but declared dependencies. For Java projects, fundamental efforts have been contributed by communities to

[1] `www.ing.com`

help developers analyze dependencies statically in JDK[2] and Maven[3]. Also, advanced tools [41, 16, 22] have been built to address prevalent dynamic language features: reflection, dynamic proxy and classloading [26]. Despite these efforts, finding unused dependencies of Java projects is not a trivial goal. It has been demonstrated that all state-of-the-art static analysis frameworks fail to capture complete dynamic language features in call graph [30].

These limitations give rise to false positives of the unused dependency detection, which is a challenge when creating tools to help developers in removing these dependencies. When they receive false alerts too often, they are likely to ignore warnings, filter alerts, or turn away from using tools altogether [23]. To encourage the usage of tools, researchers start to think that it is not only beneficial to pursue the precision of tools but also critical for tool authors to present warnings from developers' perspectives [32]. Specifically, we want to improve developers' comprehension of how unused dependencies are identified by static analysis tools.

The underlying principle of current static dependency analysis tools is reachability. Any dependency is classified as unnecessary if it cannot be reached from the application code. Based on this binary result i.e., reachable or unreachable, tools provide recommendations that help developers remove unused dependencies. However, we argue that the existing analysis result produced by tools shows more information than a binary recommendation. If we interpret the reachability in more detail, for example, complexity and evolution of method calls between artifacts, tools may report the reasoning behind recommendations and allow developers to be effective in their decision-making.

Making a decision on removing a dependency is far from an easy task, especially for software in production. Developers need to balance the potential risk of making serious errors and the future benefit of saving maintenance efforts. In this regard, the reasoning behind such a decision should be adequate to minimize the potential risk; otherwise, developers will not take risks to reduce maintenance. Thus, we want to investigate which process would enable developers to make such a decision. In other words, we examine several aspects of the results generated by static analysis tools and build a decision framework to illustrate the process of decision making. As developers are given more information about the unused dependencies, they have more evidence to support their last call.

To achieve this goal, we examined one web application at ING that had been actively maintained in production for more than a decade. Since it had been developed for a long time, we believed that there must be some unnecessary dependencies declared in the project. These unnecessary dependencies may be declared by previous developers but ignored by people who took over the job later. But before designing a framework that may help developers decide whether a dependency can be removed, it is ideal for us to identify all the unused dependencies, point out the limitation of the state-of-the-art tool, and propose a solution to fill the gap. However, the trickiest part of this research is that we may never obtain the complete ground truth of which dependencies were indeed unnecessary. It was not only because false positives of unused dependencies were unavoidable by current software techniques, but also because there were so many dependencies in the legacy code that

[2] https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

[3] https://maven.apache.org/shared/maven-dependency-analyzer/

developers may not even fully comprehend.

Although the ground truth of whole unused dependencies are not available, they can still be identified in a gradual manner: use the state-of-the-art tool to capture potentially unused dependencies, interview developers to pick out false positives based on their experiences, validate the candidates of unused dependencies by system and functionality tests, and reiterate these processes to finalize the list of unused dependencies. Once the analysis result is derived, it may help us design and verify a decision framework for recommending the unused dependencies.

The results of this thesis show that two main approaches can reduce false positives of unused dependencies. One approach is to increase the soundness of the dependency analysis by different call graph tools; the other approach is to observe the history change of method calls between dependencies. In addition to reducing the number of false positives, our decision framework also helps developers prioritize which unused dependencies can be removed first and which unused dependencies may take more effort to investigate. Moreover, a dependency visualization approach is also proposed to assist developers to comprehend and explore the unused dependencies in the call graph.

For the remaining part of this thesis, the company ING bank and the contributions this thesis makes are explained. Next, the state-of-the-art dependency analysis tool, the structure of the given web application at ING, and a motivating example that gives a clear direction to this thesis are further discussed. The chapter after described related work. This chapter was followed by a description of the static call graph construction tools and software metrics for unused dependency. In the chapter after that, our decision framework was presented in detail. The following two chapters evaluate the decision framework via the ING project and open-source projects. Finally, we draw the main conclusions and share our future work.

1.1 ING Bank

ING aims to make banking frictionless in a continually changing world. In this era when tech companies like Google, Facebook, and Amazon keep bringing new experiences and possibilities for customers, ING strives for serving their customers on these digital platforms using new technologies. Vantage is an ING's inhouse-developed platform that multiple internal applications and teams frequently apply to deliver business value. This platform was first built a decade ago and was gradually migrated into modern architecture to meet customers' demands. Since the platform team only wants to migrate functionalities or dependencies that are being used, this thesis aims to find unused dependencies in Vantage, so the platform team doesn't need to migrate them.

The journey of migration is planned to take multiple years because there are hundreds of components developed in the past decades. These components are shared as tools, web applications, libraries, solutions, or plugins by a large number of EAR files which are deployed on a self-possessed IBM WebSphere server. In this thesis, we focused on analyzing one EAR file which packages 144 components via Maven as their dependencies. After finding unused dependencies in this EAR file, the corresponding dependencies declared in the POM file are removed.

1.2 Contributions

This thesis makes the following contributions:

- Combine call graphs built by different tools to enhance the soundness in the aspect of dynamic features.
- Develop a definition to classify dependencies based on a call graph and then design a decision process accordingly to examine the history change of the relationship between dependencies.
- Apply these two approaches to filter out more than one-third of false positives of detected unused dependencies in a project at ING. Also, developers may prioritize which dependencies to be removed according to the recommendation of the decision process.
- Establish a process to visualize the relationship between dependencies in a call graph.

Chapter 2

Background

This chapter aims to explain how existing static dependency analysis tools are insufficient to identify unnecessary dependencies. Firstly, the limitation of static dependency analysis is discussed, so we can know how the analysis result is affected. Secondly, a state-of-the-art tool, DepClean is selected to exemplify how a static dependency analysis tool works. Next, an Enterprise User Management Application at ING to be analyzed is presented. Last, the result of analyzing ING's application and the result of analyzing three open-source projects by DepClean are given as motivating examples for this thesis.

2.1 Dependency analysis

To discuss the limitation of static dependency analysis, the difference between static and dynamic dependency analysis is firstly introduced. Next, soundness and precision are defined to represent the limitation, which is caused by dynamic features and call graph algorithms. Last, the limitation of static dependency analysis on false positive and false negative is summarized.

2.1.1 Static vs. dynamic dependency analysis

Dependency describes relationships between an application software and its code library while dependency analysis represents the process of finding these relationships[1]. Dependency analysis can be conducted in either a static or a dynamic manner. For the static dependency analysis, a call graph of all the dependencies has to be built and a set of entry methods have to be given. Based on the call graph and entry methods, all of the accessible classes can be found iteratively and applied to determine which dependency is used or not. However, the main limitation of the static manner is the capability of detecting method calls invoked by reflection, dynamic proxy, and callbacks from native code. These dynamic features are prevalent and difficult for a static dependency analysis tool to accurately model.

Unlike the static manner, the dynamic dependency analysis tool does not have this limitation. For the dynamic manner, the behavior of the system is monitored by logging every method of entry and exit. There are several tools that can be used to manipulate bytecode for instrumenting the code of monitoring. For example, there are low-level tools such as

the ASM library and high-level frameworks such as Javassist. Whenever the method is triggered by test cases or run-time execution, all the traces are collected and saved in the log files for building a directed call graph. Nevertheless, there are also a few limitations in the dynamic dependency analysis. For instance, it takes a long time to get all possible traces from run-time execution. Also, the coverage of built-in test cases is limited, and the performance of the system may be affected due to excessive trace collection. Hence, this thesis focuses on the ***static dependency analysis*** because the financial application is critical. By using the static dependency analysis, the existing services in the production will not be affected.

2.1.2 Soundness vs. Precision

Soundness is defined to evaluate what is the percentage of method calls invoked at run time can be found in the call graph, while precision is to evaluate what is the percentage of method calls in a call graph that is actually invoked at run time. These definitions of soundness and precision are illustrated in Fig. 2.1. The requirement of soundness and precision may differ based on the applications.

For the static dependency analysis, both soundness and precision may have implications for the results. If the soundness is low, many of the method calls invoked at run-time are missing in the call graph. Hence, some used dependencies would be considered unused by the dependency analysis tool because those dependencies are not accessible from the application code. In other words, low soundness would cause ***false positives***.

In contrast, if the soundness is high but the precision is low, the call graph contains many redundant routes that are never accessed at run time. Hence, some unused dependencies would be considered as used by the dependency analysis tool because those dependencies are accessible from the application code by the redundant routes. Specifically, low precision would cause ***false negatives***.

The limitation of the soundness is mainly caused by the dynamic features since the call graph tool fails to capture the method calls invoked by the dynamic features at run time. On the other hand, the limitation of the precision is mainly caused by the overestimation of the call graph algorithm such as Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA).

2.1.3 Dynamic features

Dynamic features of Java include reflection, dynamic proxies, invokedynamic and so on [39]. These features are ubiquitous and are the main cause of the unsoundness. Generally, dynamic features are designed for a program to examine and modify its execution state at run time [26]. Since the behavior of dynamic features can only be observed at run time, it poses a great challenge for the static analysis tools. For example, when a Java class is accessed via Class.forName, the string of the class name must be calculated at run time. To access the string name statically, the static analysis needs to be performed conservatively. In other words, we need to assume that all classes can be accessed as a candidate for Class.forName.

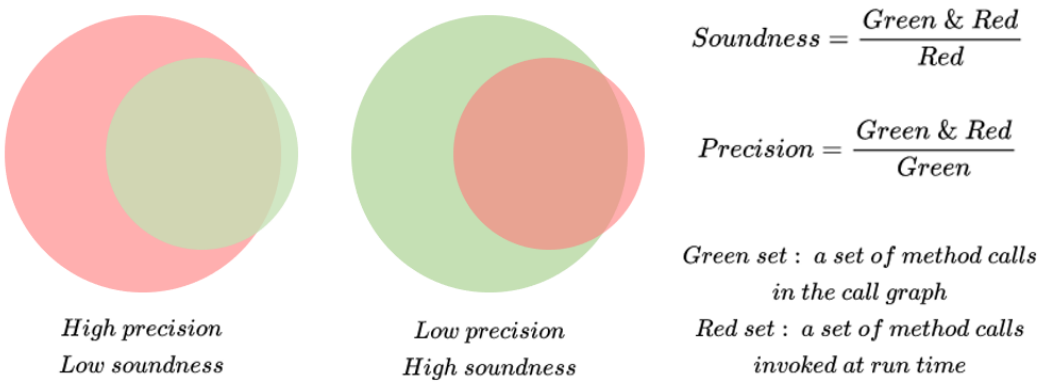


Figure 2.1: Definition of the soundness and precision

However, this approach creates a problem that the edges in the call graph would be over-estimated [34]. By accepting all the possible call sites, the precision is compromised.

2.1.4 Call graph algorithms

There are three representative algorithms for the call graph construction. These algorithms differ in their tradeoffs between soundness and precision. These algorithms are Reachability analysis (RA), Class Hierarchy Analysis (CHA), and Rapid Type Analysis (RTA). Although there are still many variants for the optimization, only these three algorithms are mentioned because the OPAL framework combined with RTA supports more dynamic features than other combinations of frameworks and algorithms [30].

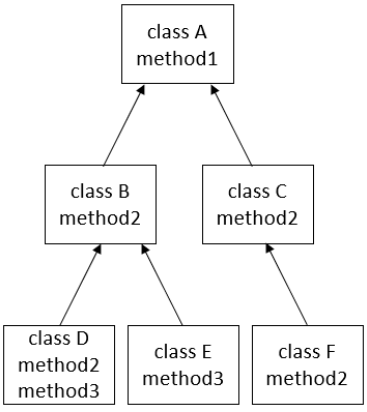


Figure 2.2: Example of the class hierarchy [24]

RA is a general idea of finding all the reachable states for a system. In the context of building a call graph, it represents a process that generates connections between the call sites and their target methods by only matching the method names. Hence, the RA call graph is conservative and its soundness must be high relative to other algorithms. Since this

algorithm searches for method calls simply based on method names, it is obvious that many of the method calls are overestimated and never be invoked at run time.

CHA aims to improve the overestimation of RA by considering the class hierarchy. The class hierarchy is retrieved from the relationship between classes by looking up the inheritance. Fig. 2.2 shows an example of the class hierarchy. By referring to the class hierarchy, we may filter out method calls that must not happen at run time. For example, if a method2 of an object with type class C is called, there are two possible callers: C.method2 and F.method2. In contrast, if the class hierarchy is not available and RA is applied, another two callers B.method2 and D.method2 would be falsely included. Hence, CHA provides better precision than RA.

RTA further improves CHA by taking into account if the class is instantiated. It is because only an instantiated object can call the target method. In other words, RTA prunes unnecessary method calls that are invoked by uninstantiated classes. For example, if class C in Fig. 2.2 is instantiated but class F is not, only C.method2 would be added to the call graph. Hence, RTA provides better precision than CHA and the call graph built by RTA is a subset of CHA as shown in Fig. 2.3.

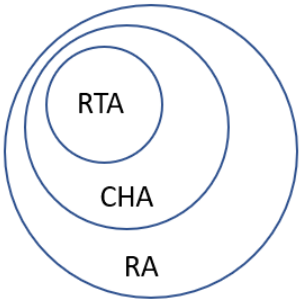


Figure 2.3: Common algorithms for the call graph construction

2.1.5 The effect of the unsoundness and imprecision

Table 2.1 summarizes the effect of the unsoundness and imprecision on the result of the static dependency analysis. To determine if the result of the static dependency analysis is correct or not, we can rely on the result of the dynamic analysis and consider it as ground truth [39]. When the static dependency analysis considers a dependency as used, it is still possible that none of the methods in this dependency is invoked at run time. It is because the call graph of the static dependency analysis is overestimated depending on the implementation of call graph algorithms (imprecision). On the other hand, when the static dependency analysis considers a dependency as unused, it is still possible that some of the methods in this dependency are invoked at run time. It is because the call graph of the static dependency analysis fails to model and detect these method calls (unsoundness). **In this work, one goal is to reduce the number of false positives (or unsoundness) so that developers may have less false alarms.**

Table 2.1: The limitation of the static dependency analysis

Static dependency analysis result	Static Analysis		Dynamic analysis (GT♯)	Outcome⋆
	Is any class in the call graph	Is any class accessible from entry classes	Is any class invoked at run time	
Used	Y	Y	Y	TN
	Y	Y	N	FN
Unused	Y	N	Y	FP
	N	N	Y	FP
	Y	N	N	TP
	N	N	N	TP

⋆TP: true positive, TN: true negative, FP: false positive, FN: false negative
♯GT: ground truth

2.2 Mechanism of a state-of-the-art dependency analysis tool - DepClean

DepClean is a state-of-the-art dependency analysis tool that extends the Maven dependency analyzer[1]. How DepClean executes the dependency analysis is exemplified in the following steps.

STEP 1 Compile the source code of a Maven project and get all the class files as entry classes. After compiling the source code by Maven command `mvn compile`, all the class files of the source code are generated in a `classes` folder as shown in Fig. 2.4.

STEP 2 Download artifacts of all the dependencies in a Maven project.
After downloading all the artifacts by Maven command
`mvn dependency:copy-dependencies -DoutputDirectory=./target/dependency`
all the artifacts are saved in a `dependency` folder as shown in Fig.2.4.

```
ccchuang@ccchuang:~/maven-test/commons-collections$ ls
CODE_OF_CONDUCT.md  DEVELOPERS-GUIDE.html  NOTICE.txt  PROPOSAL.html  RELEASE-NOTES.txt  src
CONTRIBUTING.md     LICENSE.txt            pom.xml     README.md      SECURITY.md        target
ccchuang@ccchuang:~/maven-test/commons-collections$ ls target/
antrun   classes              dependency          generated-test-sources  rat.txt       tree.txt
apidocs  depclean-results.json  generated-sources  maven-status            test-classes
```

Figure 2.4: Folders of the dependency analysis

STEP 3 Decompress artifacts of all the dependencies and create a
`DependencyMap<dependency, Set<ClassName>>`
The `DependencyMap` is used to find the corresponding dependency for each class.

[1] `https://maven.apache.org/shared/maven-dependency-analyzer/`

STEP 4 Build a call graph of all entry class and dependency class files.
The call graph is a directed graph which is built by JGraphT library.
(`org.jgrapht.graph.DefaultDirectedGraph`)

STEP 5 Traverse the call graph from entry classes to find all the accessible classes.
(`org.jgrapht.traverse.DepthFirstIterator`)

STEP 6 Search the DependencyMap for each accessible class and find the corresponding dependency.

STEP 7 Examine the usage of class files in every dependency.
Fig. 2.5 is an example of the dependency analysis result. The analyzed Maven project is `org.apache.commons.commons-collections4`.
The project declares 14 dependencies and Fig. 2.5 shows the Maven dependency tree of the project. Any class which is accessible from the entry classes is labeled in blue color; otherwise, labeled in red color. If any class in a dependency is labeled by blue color, the dependency is considered as used. Table. 2.2 summarizes the usage of all the declared dependencies in this Maven example.

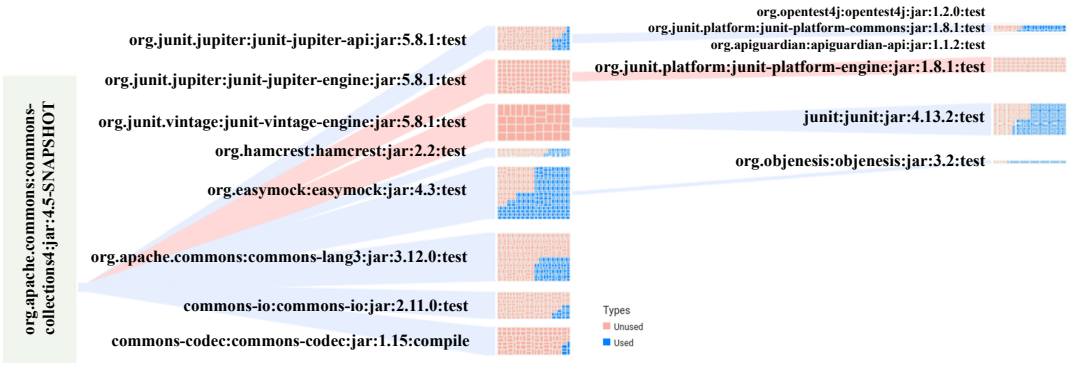


Figure 2.5: Example of the dependency analysis by DepClean

Table 2.2: The example of the dependency analysis by DepClean

Example of a Dependency Analysis Result By DepClean	
Used dependency (11)	org.junit.jupiter:junit-jupiter-api:jar:5.8.1:test org.hamcrest:hamcrest:jar:2.2:test org.easymock:easymock:jar:4.3:test org.apache.commons:commons-lang3:jar:3.12.0:test commons-io:commons-io:jar:2.11.0:test commons-codec:commons-codec:jar:1.15:compile org.opentest4j:opentest4j:jar:1.2.0:test org.junit.platform:junit-platform-commons:jar:1.8.1:test org.apiguardian:apiguardian-api:jar:1.1.2:test junit:junit:jar:4.13.2:test org.objenesis:objenesis:jar:3.2:test
Potentially unused dependency (3)	org.junit.jupiter:junit-jupiter-engine:jar:5.8.1:test org.junit.vintage:junit-vintage-engine:jar:5.8.1:test org.junit.platform:junit-platform-engine:jar:1.8.1:test

2.3 An enterprise user management application at ING

The enterprise user management application is an EAR file (enterprise application) that runs on an IBM WebSphere Application Server. The EAR file is composed of web application archives (WAR) files, enterprise beans Java archive (JAR) files, and configurations. Since the EAR file is developed and managed by Maven, all the information of JAR files, WAR files, and their dependencies are declared in a Maven Project Object Model (POM) file. Generally, every software release includes an EAR file and a POM file, with which the dependency analysis is conducted.

The Maven project at ING has a distinct structure compared to generic Maven projects. Normally, a web application's dependency tree has a similar structure in Fig. 2.6a. Given an EAR file and a POM file, the dependency analysis tool can identify how many dependencies are assembled and how a deployable EAR file is built by multi-module Maven projects. Nevertheless, developers of the provided Maven project at ING conventionally exclude all the transitive dependencies of direct dependencies and manually migrate them to a dependency-specific module as shown in Fig. 2.6b. The purpose of this extra step is to fix versions of dependencies and prevent unsafe dependency updates. These updates that may include unexpected vulnerable functionalities without notice [21]. Since transitive dependencies are excluded, the provided POM file can not help us resolve the original Maven dependency tree of a web application module.

Since the state-of-the-art dependency tool is designed for generic Maven projects, we have to transform the structure of the Maven project at ING before applying the dependency analysis tool. To resolve excluded transitive dependencies of the POM file in release, it is necessary to retrieve the original POM file in the repository of direct dependencies. It is because all the excluded transitive dependencies are declared in the POM file of every direct

dependency. After obtaining POM files of all the declared dependencies, all the excluded transitive dependencies can be remapped to the corresponding direct dependency like in Fig. 2.6c. By doing so, we can apply the dependency analysis tool to analyze ING's project.

2.4 Motivating example: analysis of applications at ING and open-source projects

As there is no static analysis tool that can guarantee its finding on unused dependencies, tools usually emphasize the result with a warning such as "potentially" unused dependencies as shown in Table 2.2. This warning could notify developers that their analysis results should be accepted with caution.

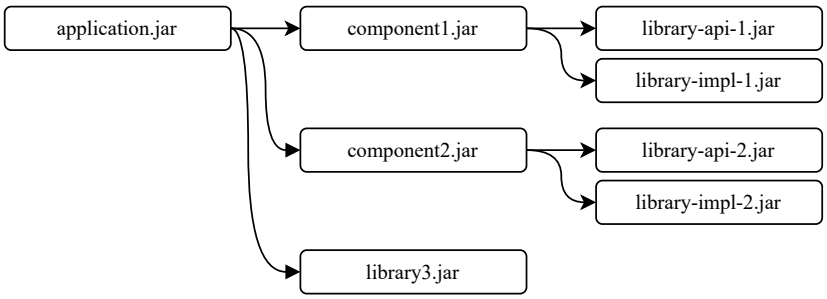
Table 2.3 summarizes the analyzed results of one ING's project and three open-source projects using the state-of-the-art tool DepClean[2]. The table presents the number of used dependencies and potentially unused dependencies. Part of these projects possesses a significant number of unused dependencies, which is unusual and unexpected. The hypothesis is that some of the potentially unused dependencies may be false positives as shown in Table 2.1 – i.e., a dependency that is being misclassified as unused. This happens, for example, in cases where it is difficult to collect using static analysis all the possible entry methods of a dependency. Hence, it may happen that all classes within the dependency become unreachable from the application, and tools misclassify it as unused.

To reduce false positives and improve the recommendation, we propose a framework that 1) combines different call graph tools, 2) considers the history of changes in the dependencies of a project, and 3) guides developers through the decision process before yielding final results. The framework will be discussed in Chapter 4.

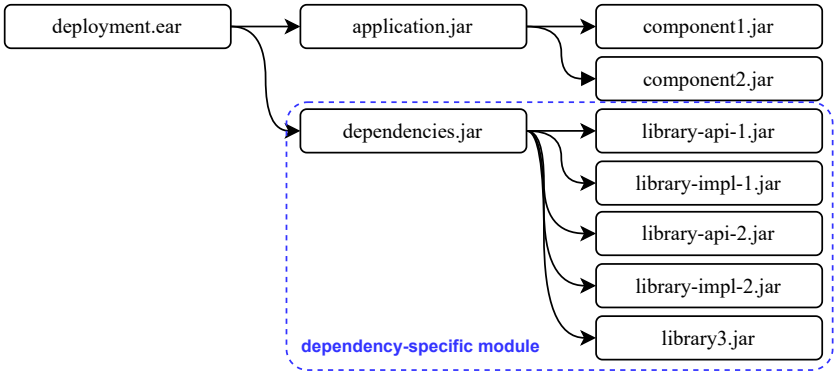
Table 2.3: The summary of dependency analysis by DepClean

Project Name	Number of Dependencies	
	Used	Potentially Unused
enterprise user management app	73	71
jenkins [core v2.343]	80	16
zipkin [zipkin-server v2.23.16]	94	22
onedev [server-core v7.0.9]	176	72

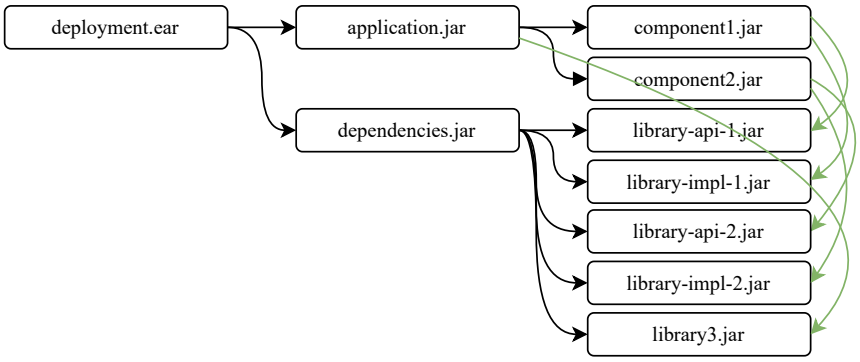
[2] https://github.com/castor-software/depclean/releases/tag/2.0.0



(a) Example of a web app's dependency tree



(b) Exclusion and migration of transitive dependencies to one module



(c) Mapping excluded dependencies to specific components

Figure 2.6: Preparation of dependencies for call graph constructions

Chapter 3

Related work

Dependency analysis is a well-explored domain in computer science. Over the past decades, precision is a primary focus of research in static analysis while soundness is attracted attention not until recently [38]. Since this work aims to improve unsoundness or reduce the number of falsely detected unused dependencies, this chapter discusses literature reviews of several dependency analysis tools, various ways related to evaluating the soundness of the call graph construction, and the influence of unsoundness for the dependency analysis. Lastly, some missing points of literature are described.

3.1 Dependency analysis tools

This section focuses on the dependency analysis tools for the Maven project of Java. First, several frameworks that are designed to remove bloated dependencies are introduced. Next, some empirical studies based on the dependency analysis tools are mentioned.

3.1.1 Dependency analysis tools for removing unused dependencies

Table 3.1: Representative dependency-debloated tools

Dependency analysis tools	Type
DepClean (2021) [37]	static
JShrink (2020) [11]	static + dynamic
JDBL (2021) [35]	dynamic

The topic of removing unused code has been explored primarily on C/C++ software projects instead of Java. Until recently, some researchers start to pay attention to Java and build frameworks to search for unused code in Java projects. Table 3.1 summarizes three representative dependency-debloated tools. DepClean takes a purely static approach; JShrink combines the static approach with the dynamic approach; JDBL is an exclusively dynamic analysis tool.

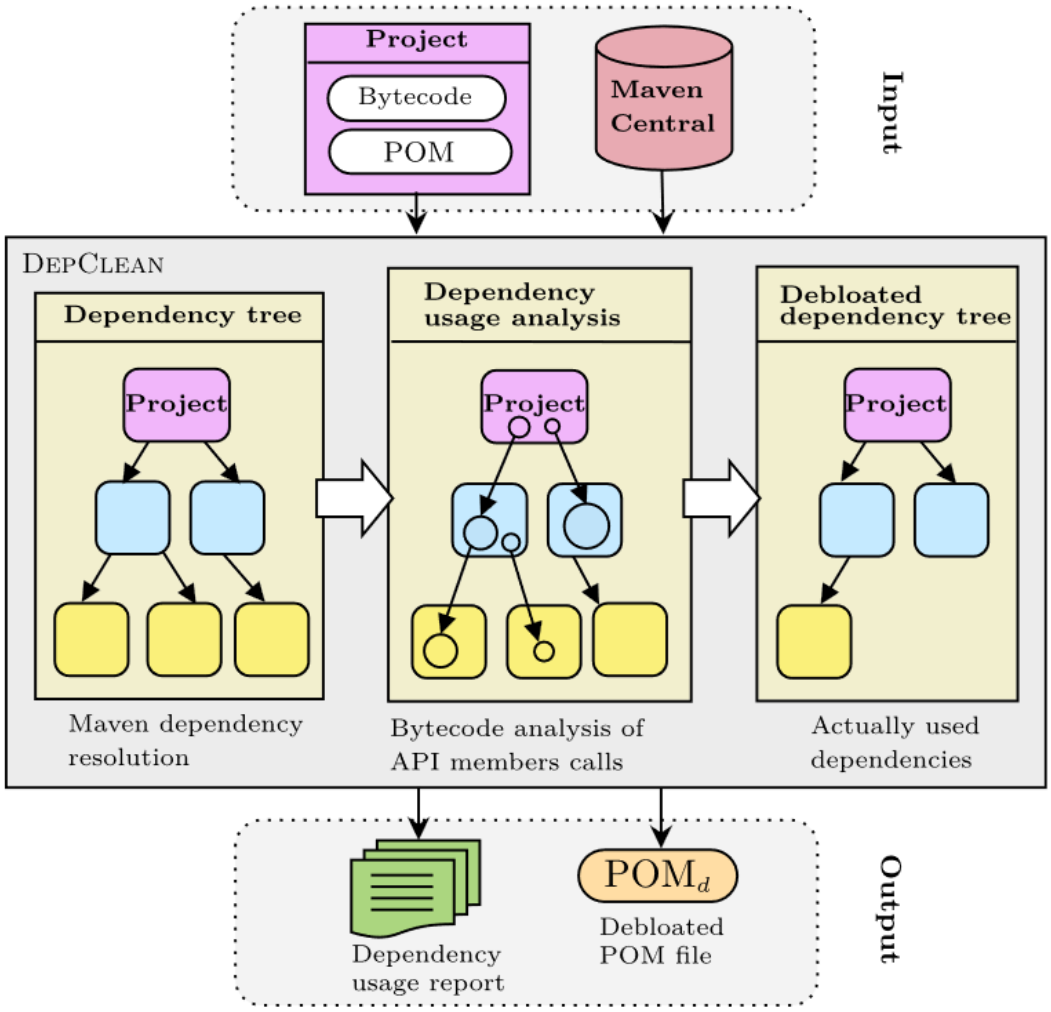


Figure 3.1: Overview of DepClean workflow [37]

The first team has conducted a study to find out the presence of unused dependencies in Maven artifacts [37]. The authors develop a tool called DepClean, which extends the maven-dependency-analyzer maintained by the Maven team. They collect a list of dependencies declared in the POM and analyze the bytecode to identify all potentially unused dependencies. They analyze the bytecode by using ASM library[1], which captures annotation, field, method, and limited dynamic features like class literals in each class. But when analyzing a dependency that is invoked by other dynamic features, a used dependency may be considered unused due to missing edges established by dynamic features.

[1] https://asm.ow2.io/

Their goal is to generate a variant of the POM without those unused dependencies. The workflow of DepClean is presented in Fig. 3.1. The inputs are the bytecode of the project and the corresponding POM file while the outputs are a dependency usage report and an updated POM file without declaring unused dependencies. DepClean requires the analyzed project as a Maven project. One reason is that DepClean uses Apache Maven API[2] to manipulate the dependency. Another reason is that the report of dependency usage follows the convention of Maven. Hence, DepClean not only tells which dependency may be unused but also mentions whether the dependency is direct or transitive.

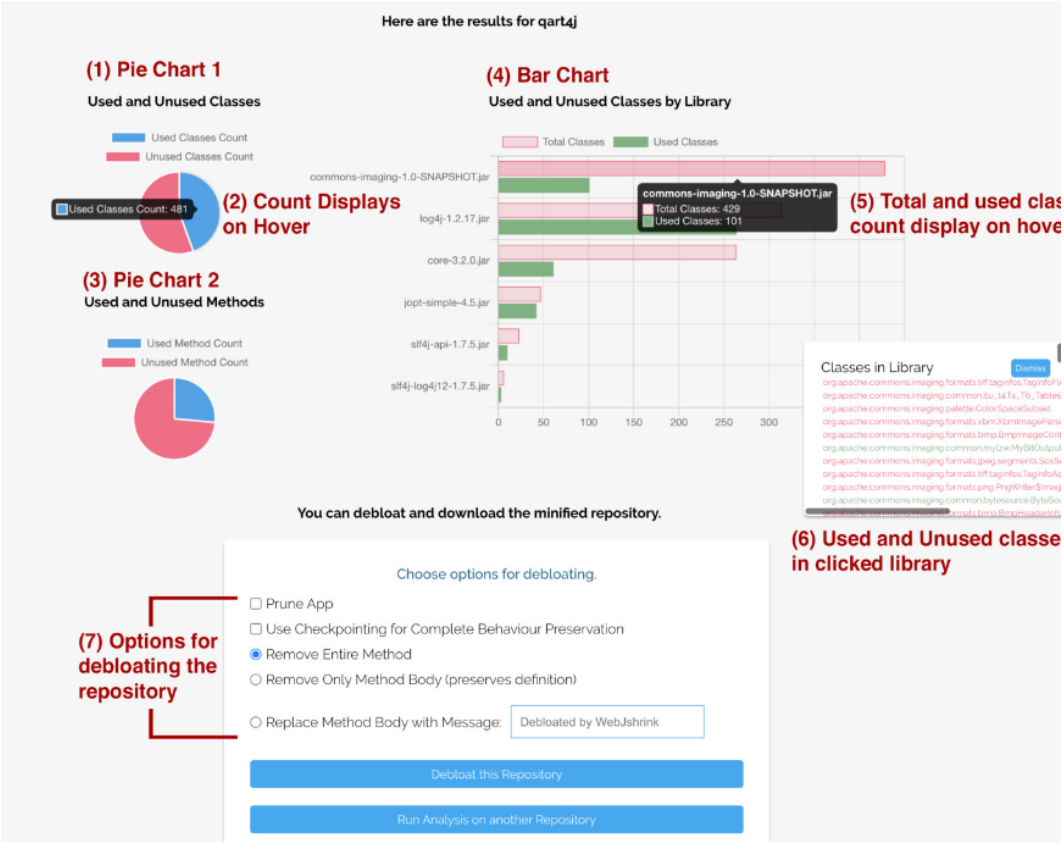


Figure 3.2: Web page of JShrink for visualization [27]

Another team has augmented the static reachability analysis with dynamic reachability analysis [11]. Their tool, JShrink, uses test cases to find dynamic features invoked at runtime and adds them back to amend the static call graph. Their analysis is fine-grained down to the level of method and field. To preserve behaviors after removing unnecessary bytecode, they build type dependency graphs using the ASM library to ensure type safety. Compared to their work, our analysis is coarse-grained at the artifact level and purely static. In our targeted scenario, test cases are not widely available or only include a few basic ones;

[2] https://maven.apache.org/ref/3.8.1/apidocs/index.html

therefore, JShrink cannot provide much help. Also, our intention is to remove unused dependencies for modules. Instead of removing redundant bytecode in each artifact, we notify developers to refactor when the usage of an artifact is relatively low.

Additionally, the authors make use of this tool as a backend of WebJShrink[3], a visualization interface allowing developers to select removal options [27] as shown in Fig. 3.2. The removal options range from aggressive ones to conservative alternatives. The most aggressive one is removing both application and library code if the library is found unnecessary. The most conservative one is adding an exception message to the place where the code is removed. Hence, if a method is removed incorrectly, an exception will be triggered at run time, and developers will know which removed code should be reverted. The visualization interface also presents the usage of dependencies in class and method levels to help developers make decisions. Similar to the design intention of this visualization tool, we acknowledge that the decision of removing dependency should be made by developers rather than the tool's authors. Inspired by the work, this thesis applies Neo4j bloom to visualize the call graph and present the visualization to developers.

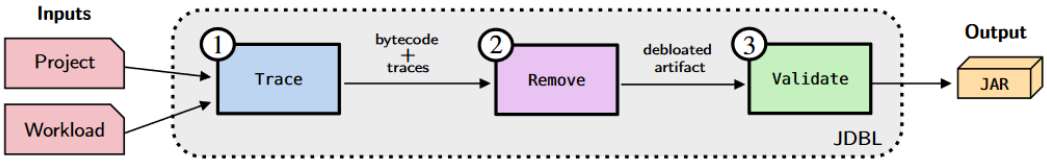


Figure 3.3: Overview of JDBL workflow [35]

The third team develops a purely dynamic dependency analysis tool Java DeBLoated (JDBL). Since the soundness of the dynamic dependency analysis depends on the coverage of the trace collection, JDBL integrates four coverage tools to collect traces: JaCoCo, JCov, Yajta, and JVM class loader. It is because coverage tools support diverse corner cases. When executing the coverage tools, traces are collected by the bytecode instrumentation. Bytecode instrumentation is an approach to monitoring the project at runtime by adding probes in the class files. Once the probe is activated at runtime, the coverage tool will report the trace. After collecting all traces, any class and method that are not activated at runtime will be considered bloated or unused.

The workflow of the JDBL has three phases and is presented in Fig. 3.3. For the Trace phase, the coverage tools are triggered by a workload which is a set of test scenarios. When the workload is replayed, the trace is collected accordingly. For the Remove phase, JDBL replaces the implementation of unused methods with exceptions like JShrink so that the program can pass validations in the next phase. For the Validate phase, the debloated program is validated by Maven Build Life Cycles and test cases of the input workload. Although this thesis does not take a dynamic approach to analyzing dependencies, the idea of applying different tools to enhance the soundness of trace collections at runtime is necessary. Inspired by this work, this thesis augments the call graph of DepClean by adding support of a static call graph tool OPAL[4] so that dynamic features of Java language can be detected.

[3] https://www.youtube.com/watch?v=yzVzcd-MJ1w

[4] https://github.com/fasten-project/fasten/tree/develop/analyzer/javacg-opal

3.1.2 A longitudinal study of the dependency bloat

The authors who develop DepClean apply their dependency analysis tool to conduct a longitudinal case study for investigating how unused dependencies increase, decrease or remain stable over time in hundreds of single-module Maven projects [36]. Motivated by the fact that bloated dependencies are prevalent in the Maven ecosystem, they want to understand when the bloated dependencies happen, how the bloated dependencies evolve, and whether bloated dependencies have an impact on maintenance.

Fig. 3.4 shows the workflow of the longitudinal case study. There are three procedures: Collect, Filter, and Analyze.

- For the Collect procedure, hundreds of single-module Maven projects using Java as the primary language are collected via GitHub API.
- For the Filter procedure, all the collected projects are sorted by the number of releases, and only the first 500 projects are selected. Once projects are selected, the procedure collects the commit of pom.xml in every release of all the filtered projects and also collects the commit of pom.xml generated automatically for the update of dependencies.
- For the Analyze procedure, the selected projects are checkouted or reverted according to the collected commits. For every checkout, the DepClean is applied to analyze the usage of dependencies. Eventually, the evolution of dependencies can be observed by the usage of the dependency tree over time.

However, in this thesis, instead of removing unused dependencies from the pom.xml based on a binary evaluation of the bytecode – i.e. used or unused, the relationship of unused dependencies with other artifacts in the call graph is provided for more reasonings.

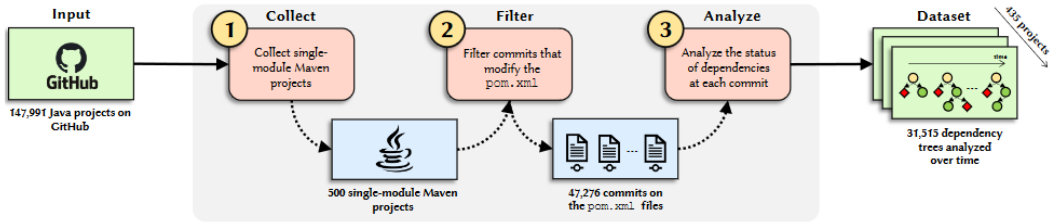


Figure 3.4: A longitudinal study of the dependency bloat by DepClean [36]

3.2 Static call graph construction tools

Call graph construction is a principal element of static analysis tools to determine unused dependencies. Previous work has laid out the difficulties of building a sound and precise call graph statically in Java. The main obstacle is posed by the usage of the Reflection API, a great mechanism for developers to inspect and adapt the behavior of their software in the runtime environment [26]. Since tools cannot correctly predict how software is evolving,

tools simply consider all the possibilities, leading to unsoundness and imprecision. According to an empirical study [25], as many as 78% of 461 representatives open-source Java projects contain at least one usage of the Reflection API. Likewise, the same study found that 21% of open-source Java projects use dynamic proxies, a proxy mechanism that brings the flexibility of forwarding method calls to different objects at runtime. This mechanism also generates a dynamic layer against static analysis tools to model [16].

Many existing static analysis frameworks have implemented numerous call graph algorithms to cope with these challenges. Previous work has conducted a comparative study of four state-of-the-art frameworks: Soot, WALA, DOOP, and OPAL [30]. Their result in Table. 3.2 helps us understand the performance of frameworks w.r.t the profile of language features supports. A circle symbol indicates whether all, some or none of the tests pass. Overall, OPAL with the Rapid Type Analysis (RTA) call graph algorithm [31] is the most feature-completed option which enables users to solve prevalent Java dynamic features and APIs among 23 categories grouped from predefined 122 test cases. OPAL is also the fastest framework due to its scalability. This scalable feature ensures that data structures are immutable while call graphs are constructed in parallel [14].

Moreover, OPAL provides an API[5] to analyze merely a portion of dependencies by classifying them as **project files** or **library files**. Since this API only captures outgoing method calls from project files to library files, it is efficient to analyze the provided enterprise user management application with a normal laptop[39].

Other studies have built reflective analysis [26] and dynamic proxy support [16] on the top of the DOOP framework with high accuracy. However, DOOP's call graph generator is so time-consuming and memory-intensive that it is impractical for real-world usage. As far as we know, no studies have discussed the performance of applying these frameworks in industrial software. In this work, the intention is to investigate how OPAL helps developers to find unused dependencies in production code.

Table 3.2: Support of dynamic features in various call graph frameworks and algorithms [30]

Category	$Soot_{CHA}$	$Soot_{RTA}$	$Soot_{VTA}$	$Soot_{SPARK}$	$WALA_{RTA}$	$WALA_{0\text{-}CFA}$	$WALA_{N\text{-}CFA}$	$WALA_{0\text{-}1\text{-}CFA}$	$OPAL_{RTA}$	$DOOP_{CI}$
CL	◐ 4/6	◐ 4/6	◐ 4/6	◐ 3/6	◐ 4/6	◐ 4/6	◐ 2/6	◐ 4/6	◐ 4/6	◐ 4/6
DP	● 1/1	● 1/1	○ 0/1	○ 0/1	● 1/1	○ 0/1	○ 0/1	○ 0/1	● 1/1	○ 0/1
J8DIM/J8SIM	◐ 3/7	◐ 3/7	◐ 3/7	◐ 3/7	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7	◐ 3/7
MR/Lamdbas	○ 1/11	○ 1/11	○ 0/11	○ 0/11	● 11/11	◐ 10/11	◐ 10/11	◐ 10/11	● 11/11	○ 1/11
JVMC	◐ 4/5	◐ 4/5	◐ 3/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5
LIB	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 2/5	○ 0/5
TR	◐ 4/9	◐ 4/9	◐ 4/9	◐ 4/9	◐ 3/9	◐ 6/9	○ 0/9	◐ 6/9	● 9/9	◐ 3/9
LRR	● 3/3	● 3/3	● 3/3	● 3/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	◐ 1/3	◐ 2/3
CSR	● 4/4	● 4/4	● 4/4	● 4/4	○ 0/4	○ 0/4	○ 0/4	○ 0/4	◐ 1/4	◐ 0/4
sum	**26/51**	**26/51**	**23/51**	**21/51**	**29/51**	**30/51**	**22/51**	**30/51**	**38/51**	**15/51**

CL: Classloading, **DP**: Dynamic proxies, **J8DIM**: Interface default methods, **J8SIM**: Static interface methods, **Lambda**: Java 8 invokedynamics, **JVMC**: JVM calls, **LIB**: Library analysis, **TR**: Trivial reflection, **LRR**: Locally resolveable reflection, **CSR**: Context-sensitive reflection.

[5] https://www.opal-project.de/library/api/SNAPSHOT/org/opalj/br/analyses/Project$.html

3.2.1 Frameworks to validate the soundness of the call graph

The soundness and precision of a call graph determine the correctness of the static dependency analysis. This section describes three representative frameworks that aim to measure the soundness of a call graph as shown in Table 3.3. Although this thesis does not pay attention to the measure of soundness, these frameworks may enhance the understanding of the cause of the unsoundness.

Table 3.3: Representative frameworks for measuring the soundness of a call graph

Framework names	Type
Recall study (2020) [39]	static + dynamic
Judge (2019) [11]	static
A benchmark and tool evaluation (2018) [38]	static

The first framework aims to measure the unsoundness of static call graph construction tools. The study uses recall as a metric to quantify the unsoundness. Since the definition of recall includes a factor of the ground truth, the authors resort to the dynamic analysis to obtain the ground truth and assume the ground truth as an oracle. Hence, the recall is measured by comparing the call graph built by static analysis tools and the call graph inferred by dynamic analysis tools. The workflow of the framework is shown in Fig. 3.5.

- **Extract test cases**. The dynamic analysis tools rely on the quality of the test cases to collect all possible traces. The higher coverage of the test cases, the better recall will be. To get higher coverage, the authors combine the built-in test cases and generated test cases. It is because many traces triggered by built-in test cases are independent of the generated test cases.

- **Unreflect test cases**. Since JUnit test cases are invoked by reflection, the static analysis tools may have difficulty detecting methods in the test cases, which leads to bias in the call graph and lowers the recall. To solve this problem, the authors design a pre-analysis to unreflect the reflection methods in JUnit test cases. In other words, all the methods using reflection API are rewritten so that they can be detected by static analysis tools. In this way, JUnit test cases can be used as static drivers to trigger the application code.

- **Static call graph (SCG)**. To build a call graph statically, the authors apply the DOOP framework which supports context sensitivity, and several dynamic language features. Although this supports offer equal or superior recall to alternative frameworks like soot and WALA, their analysis of performance indicates that it does not apply to large software projects. Among 31 dependencies, 11 of them fail the call graph construction due to timeout after 6 hours. In addition, the heap size of the JVM has to be set to 384GB for the static analyses, which prevents the framework from scaling.

- **Context call tree (CCT)**. The context call tree is a model of the call graph constructed by the dynamic analysis. The node of the tree represents a method and the edge

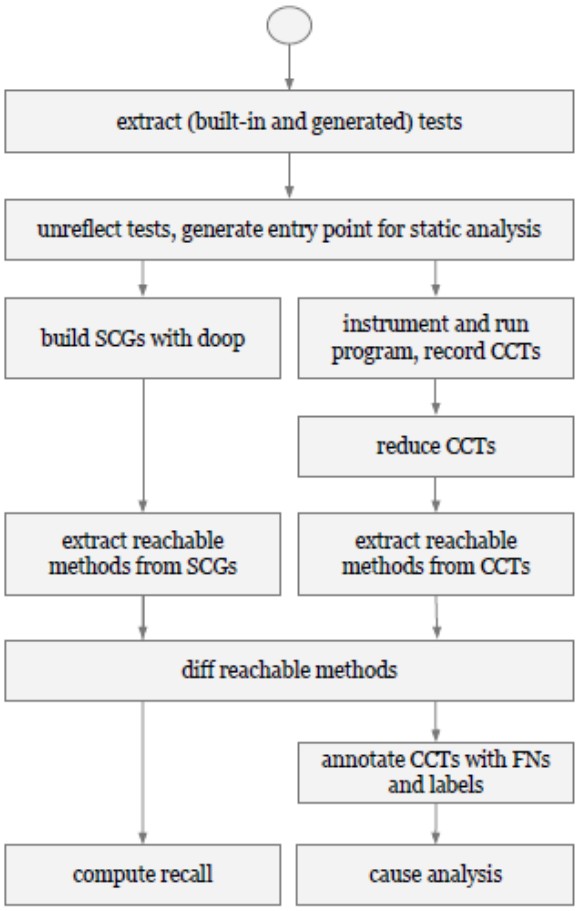


Figure 3.5: Workflow of the recall study. (SCG: static call graph, CCT: context call tree, FN: false negative) [39]

represents the method invocation while the root represents the entry method. The node and edge are observed by monitoring the runtime execution. Before executing the software by test cases, the authors apply the ASM library to instrument the code on method entry and exit. These instrumentations are logged and provided to model the oracle.

- **Compute recall**. Recall is defined in the equation 3.1. The recall is proportional to the method (`M`) coverage of the static call graph. The method set of the context call tree (`CCT`) is considered an oracle even though its coverage is unlikely to be complete except for analyzing trivial software.

$$recall = \frac{M_{SCG} \cap M_{CCT}}{M_{CCT}} \tag{3.1}$$

The second framework also aims to understand the unsoundness of static call graph construction tools. The study develops a toolchain that analyzes the call graph algorithms by test cases profiling and uses the profiles to identify the unsoundness in the call graph. The toolchain has two parallel pipelines as shown in Fig. 3.6.

The upper pipeline takes 122 test cases (grouped in 23 categories) as inputs and applies them to various call graph algorithms offered by the frameworks: Soot, WALA, DOOP, and OPAL. The profile is obtained by comparing the call graph edges and annotation in the test cases. If the annotated method of a test case has the corresponding edge in the call graph, it means that the language feature of the test case is supported. The output of the upper pipeline is a profile about which language features are supported or not by the corresponding framework and algorithm.

The input of the lower pipeline is a project bytecode for the investigation of the unsoundness. The lower pipeline has two parallel processes: **Hermes** and **call graph computation**. Hermes is a framework for extracting different features in a Java bytecode [29]. The Judge uses Hermes to find whether a project bytecode contains a specific feature and where the feature locates. Specifically, Judge applies Hermes to identify all of the 23 categorized features in a given Java bytecode. Next, the location of these features is compared with the call graph to investigate the unsoundness.

An example of investigating unsoundness is presented in Fig. 3.7. There are 6 columns in this table. The first and second columns are a profile of a specific call graph algorithm obtained by the upper pipeline in Fig. 3.6. The third to sixth columns are the mapping between the algorithm profile and the features obtained by Hermes in a call graph. Hence, we can know if any method in the call graph has a feature that the algorithm has difficulty capturing like `my` in Fig. 3.7.

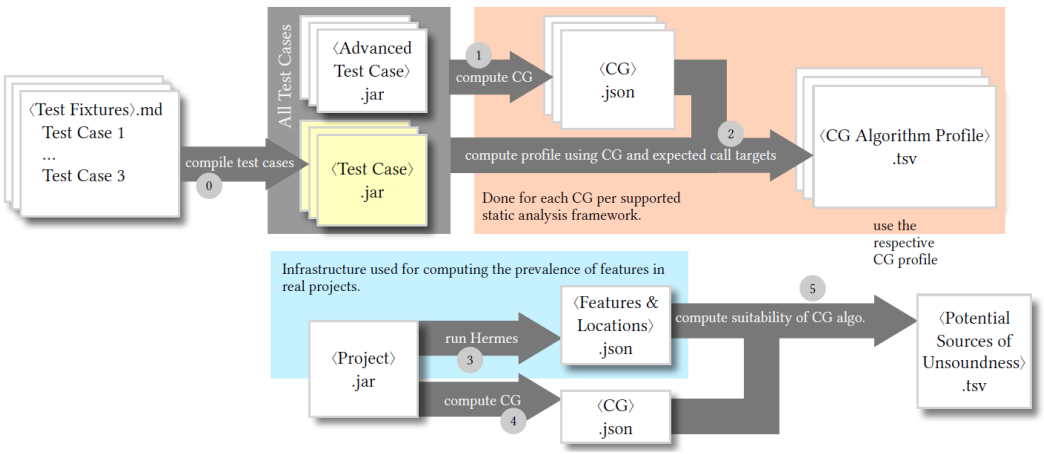


Figure 3.6: Call graph analysis toolchain -— Judge [30]

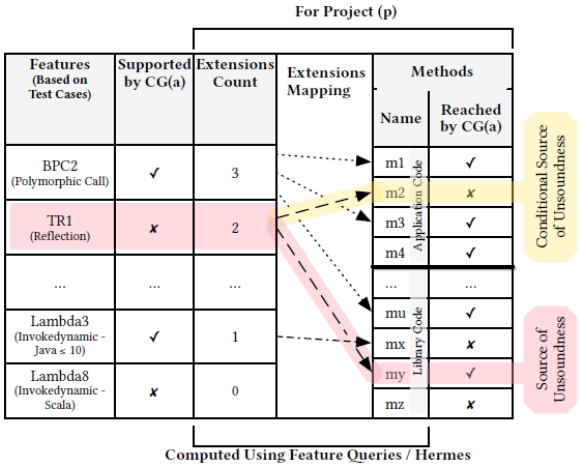


Figure 3.7: Mapping between the algorithm profile and the call graph [30]

The third framework proposes a micro-benchmark to evaluate the recall of static call graph construction tools. The micro-benchmark is a set of small Java programs, and each of them uses a specific dynamic feature that poses challenges for static analysis tools to model. The dynamic features include reflection, dynamic proxies, dynamic class loading, invokedynamic, and serialization. These small Java programs are then used to build call graphs by soot, WALA, and DOOP for evaluation.

A benchmark example of invocation is presented in List 3.1. The annotated source (i.e. `@Source`) represents the entry method of the benchmark, which can be triggered by an outside client. The annotated target (i.e. `@Target`) indicates which target method is expected or unexpected by using values such as `Expected.YES` or `Expected.NO`. An oracle of the benchmark can be generated by getting these annotated values at runtime. To evaluate a call graph construction tool, a generated oracle of a specific benchmark is compared to a call graph of the same benchmark constructed by the tool. Finally, the recall of the static call graph to the oracle can be computed.

```
public class Basic {
    public boolean TARGET = false;
    public boolean TARGET2 = false;
    @Source public void source() throws Exception {
        Method m = Basic.class.getDeclaredMethod("target", null);
        m.invoke(this, null);
    }
    @Target(expectation = Expected.YES) public void target() {
        this.TARGET = true;
    }
    @Target(expectation = Expected.NO) public void target2() {
        this.TARGET2 = true;
    }
}
```

Listing 3.1: Example of basic invocation [38]

3.3 Software metrics for unused dependency

A software metric is a common data analysis technique to support decision-making during the software development [15]. For instance, software defect analysis tools identify a set of features to help detect false alarms. The features computed from code analysis and warning history are considered as metrics [42]. These metrics may help developers understand the detailed information of false alarms, prioritize them with strategies, and prune them by heuristic rules [13]. Unlike software fault detection, there are only a few previous works on designing metrics for unused code detection. Previous work investigated whether code stability and code centrality indicate the likelihood of unnecessary code. The authors reported that 34% of recommendations for unnecessary code were confirmed as true positives [19]. Another work tried widely accepted objected-oriented metrics to predict dead code methods. It concluded that LOC, WMC, and RFC are useful indicators to discover dead code [33]. However, these previous works focus on the code level instead of the dependency level. In this work, our approach is inspired by software metrics and aims to contribute to the classification of the unused dependency for decision making.

Moreover, previous studies on unused code detection mainly paid attention to the code structure and history. On a larger scope, code dependency between entities in software can be broadly defined by various coupling, which may occur among software modules, classes or objects [9]. Systematic classification of such relations was provided by Fregnan et al. [17]. They categorized code dependency proposed by communities over the years into four groups: structural, dynamic, semantic and logical coupling. Prior research has verified that these coupling relations are orthogonal and able to explore different aspects of a software [18] [8] [7]. Likewise, Tàrrega NB et al. [40] extended several coupling metrics to measure the degree of software dependency. However, these coupling metrics have not been applied to detect unused dependencies.

3.4 What is missing?

As can be seen from the literature about dependency analysis tools in Table 3.1, various types of tools have been proposed to identify bloated dependencies. However, there are two missing points listed as follows.

- None of these tools can guarantee their findings on bloated dependencies due to the unsoundness and the imprecision of the call graph construction. Hence, it would be helpful if a tool reports to developers which bloated dependencies can be deleted with high or low confidence.

- All of these tools merely provide binary recommendations of the dependency usage i.e. used or unused, but the relationships between dependencies in the call graph are lacking. However, the history change of these relationships may reveal the dependency usages that dependency analysis tools do not pay attention to. Although a tool like JDBL considers the history change of every dependency usage, it does not consider the history change of the relationships between dependencies.

- Although DepClean supports the visualization of dependency analysis results, its dependency tree becomes ambiguous once it grows larger. User-friendly visualization of dependencies and call graphs can help developers explore the usage of dependencies.

In a similar sense, many frameworks in Table. 3.3 have been proposed to evaluate the unsoundness of a call graph. However, there are still two missing points listed as follows.

- All of the frameworks conform that unsoundness is unavoidable in a call graph built by current call graph tools and algorithms either statically or dynamically. This may be the reason why no study applies dependency analysis tools to an industrial application.
- Although OPAL supports more dynamic features than other call graph construction frameworks, there are no dependency analysis tools that adapt OPAL to analyze dependencies.

In this thesis, we propose a decision framework that adapts OPAL to support dynamic features when building a call graph. The call graph will be used to examine the history change in relationships between dependencies. This information may help distinguish which dependencies may be removed with high confidence. Unlike dependency analysis tools discussed previously, the proposed approach in this thesis tries to reduce the workload of developers by telling them which dependencies can be removed with ease and which dependencies may need more effort. Also, a visualization tool is proposed to assist developers in exploring the usage of dependencies. The work in this thesis focuses on the following aspects:

(1) Adapt OPAL, a call graph construction tool, to DepClean, a dependency analysis tool so that the augmented call graph supports dynamic features.

(2) Examine the history change of relationships between dependencies which may strengthen or undermine the findings of the dependency analysis.

(3) Propose a visualization approach to examine the unused dependencies along with their position in the dependency tree and call graph.

(4) Apply the decision framework to analyze a real-world industrial application with large amount of dependencies.

(5) Reduce false positives of unused dependencies and identify which dependencies are difficult for dependency analysis tools to capture.

Chapter 4

Framework

The decision framework for reducing false positives of unused dependencies is explained in this chapter. The ideas behind the decision framework are increasing the soundness via different call graph tools and observing the history change of method calls between dependencies. The decision framework comprises three steps to find unused dependencies:

A. **augmented CG.** Combine different call graph tools for the analysis of dependency usage to enhance soundness.

B. **graph analysis.** Classify flagged dependencies by their relationships with other dependencies in the call graph.

C. **release history analysis.** Analyze the history of code changes related to a flagged unused dependency.

These steps are pinpointed as shown in Fig. 4.1. Rounded rectangles represent procedures for every software release. First, we build an augmented call graph by which the dependency analysis flags dependencies as used and unused. Next, we classify these flagged dependencies according to their relationships in the call graph. Last, we apply the decision process to provide recommendations for the developers.

4.1 Dependency analysis based on a call graph built by different tools

Software projects may contain different language features and APIs which affects the performance of the dependency analysis tools. Hence, the framework starts by analyzing the project with DepClean to collect a preliminary result. To enhance the support of dynamic features for the static dependency analysis, the framework augments the call graph of DepClean with critical edges collected with OPAL+RTA.

Although OPAL$_{RTA}$ supports many dynamic features, the way of applying OPAL API to build a call graph has a great impact on the precision of the call graph. It is because OPAL has high coverage and is designed to find all the possible implementations of an

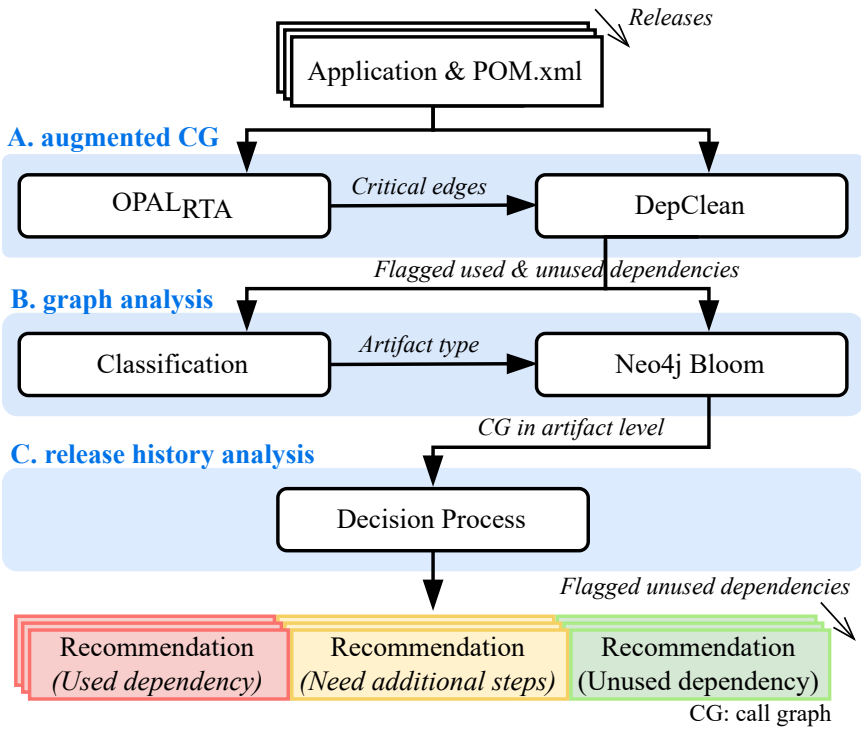


Figure 4.1: Decision framework workflow

interface or abstract class. If a global call graph is built by joining all dependencies with the main application code, there might be some spurious edges created between unrelated dependencies. To avoid the spurious edges, two factors are considered to maintain the precision of the call graph: **dependency tree**, and **critical edges**.

The framework uses the **dependency tree** to enhance the precision of generating the global call graph. It is because a global call graph might lead to theoretical edges that are inconsistent with the hierarchy defined by the dependency tree. Fig. 4.2a illustrates this process with a dependency tree of a Maven application. These dependencies are downloaded and grouped into multiple folders based on their layers in the Maven dependency tree. Next, the artifact of a parent dependency in each folder is classified as a **project file** whereas artifacts of all the child dependencies are classified as **library files** as shown in Fig. 4.3. After the classification, OPAL API is applied to build a call graph per folder. In this way, the framework avoids inconsistent theoretical edges such as a method call from `component1` to `library-impl-2`.

Critical edges are a set of edges that must be called at runtime if they are reachable from the application code. Conversely, some edges occur frequently but may not necessarily be called at runtime. For example, some methods such as toString, hasNext, or toArray defined in JDK are implemented by so many dependencies, which makes it difficult to anticipate which implementation will be executed during runtime. Hence, OPAL will create

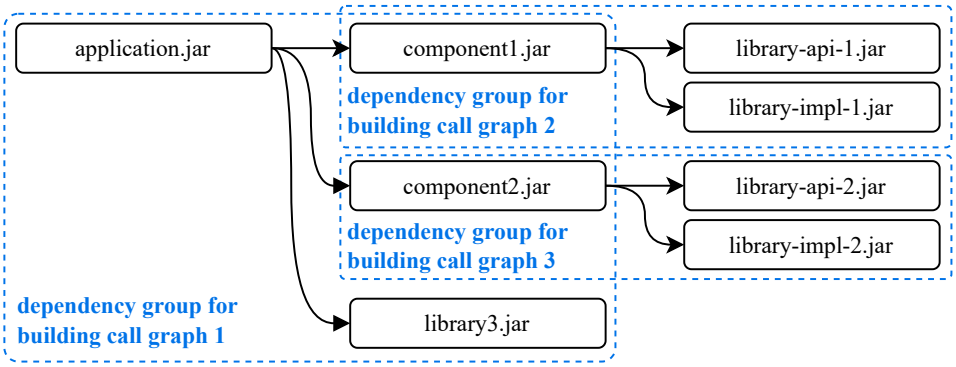


Figure 4.2: A dependency tree example.

(there may be more than 3 layers in real scenarios)

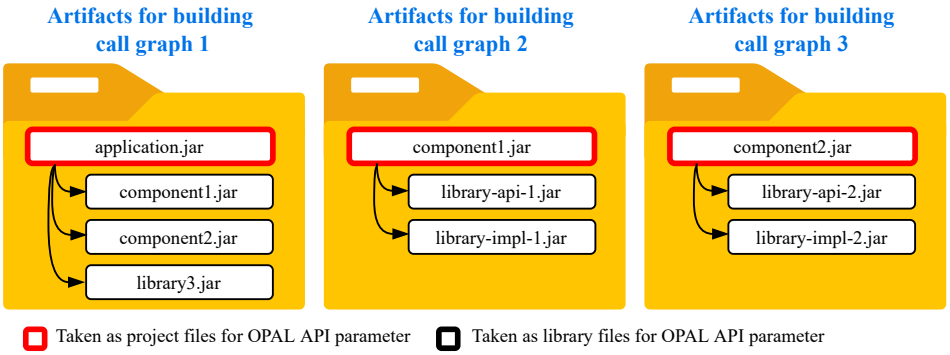


Figure 4.3: The process of separating dependencies for the OPAL call graph construction

edges leading to certain dependencies simply because they implement these methods. To prevent the call graph from exploding the framework applies the following approximation: the framework removes any edges found by OPAL that point to multiple implementations of the same method. Hence, the framework only keeps the critical edges. Since these critical edges are a unique path between a component and a target method, the framework is certain that the respective component will always call the target method during runtime. After collecting these critical edges from all OPAL call graphs, the framework adds them to the call graph of DepClean. With this approximation, the framework avoids an overly complex call graph that takes too much time to generate and analyze.

Figure 4.4 illustrates the principle of how to find critical edges. In Fig. 4.4(a), if a target method of a node (T_A) is only called by a source method of a node (S_A), this edge is considered a critical edge. It is because only T_A implements this target method and it must be called at runtime if S_A is reachable from the application code. In the case of Fig. 4.4(b), if all the nodes (S_A to S_N) merely call an identical target method once, these edges are considered critical edges. It is because only this target class implements the method. On the contrary, all edges in Fig 4.4(c) are pruned from the OPAL's call graph. It is because a target method is implemented by multiple different target classes. Since there is no clue

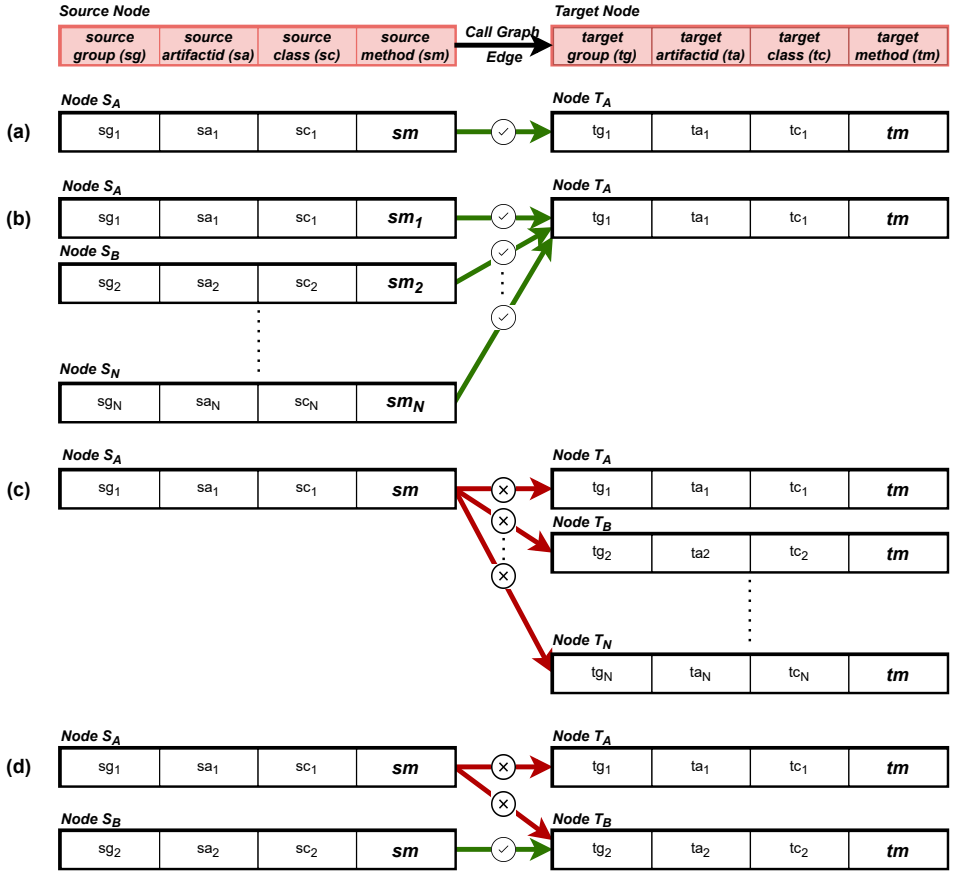


Figure 4.4: Illustration of how to find critical edges.

which implementation will be called at runtime, the framework opts to prune these edges. Hence, if previous rules are applied to Fig 4.4(d), two edges from S_A are pruned while one edge from S_B is considered a critical edge.

To flag which dependency is used or not, the framework follows DepClean's approach. A set of entry classes in a call graph must be defined. For the enterprise user management application at ING, all of the classes that handle requests are possible entries. For the open-source projects, all of the classes in the source folder are used as entries. Next, entry classes are used to traverse a call graph and find all the reachable classes. If any class of a dependency is found to be accessible from entry classes, this dependency is flagged as used. Also, as long as a class contains a method accessible from entry classes, all the methods in the class are flagged as used too. An example of flagging a dependency as used or unused is presented in Fig. 4.5.

4.2 Classification of flagged dependencies

A call graph may expand with increased dependencies and become difficult to trace, but method calls between dependencies can be simplified according to their source and target methods as exemplified in Fig. 4.5. If a method can not find a route back to any entry class or used method, the method is defined to be an unused method. An unused method could exist both in flagged used and unused dependencies.

Based on the type of incoming and outgoing method calls, a flagged dependency can be classified into five types. If the dependency can be classified as more than one artifact type, we choose the one with the largest number.

- Artifact type 1 represents an isolated dependency and it has no external method call.
- Artifact type 2 indicates that a flagged unused dependency has incoming or outgoing method calls to or from other flagged unused dependencies.
- Artifact type 3 depicts that a flagged unused dependency has outgoing method calls to any flagged used dependency.
- Artifact type 4 portrays that a flagged unused dependency has incoming unused method calls from any flagged used dependency. The incoming unused method calls are not accessible from any entry classes.
- Artifact type 5 describes a flagged used dependency.

Neo4j Bloom [4] is adapted to visualize classified artifact types and their relationship to other artifacts. In Neo4j, a graph data structure is organized as nodes, relationships, and properties. The discrete nodes are connected by relationships and both of them can be described further by properties [28]. The definition of Fig. 4.5 is followed to set properties of nodes and relationships. For example, the artifact type is one of the properties of nodes while the method call type is one of the properties of relationships. The visualization gives an overview of which dependency is flagged unused while artifact types indicate their relationships to other dependencies. This approach may help developers comprehend the detailed information of flagged unused dependencies.

Figure. 4.6 demonstrates how to use the artifact classification and Neo4j Bloom to observe the usage of one flagged dependency in the Jenkins project. In earlier versions 2.287 and 2.291, this flagged dependency is classified as artifact type 5. However, in the later version since 2.296, the flagged dependency is classified as artifact type 1. It shows that incoming and outgoing method calls of this flagged dependency decrease, which may be evidence of true unused dependency. This way, the complexity of the call graph is simplified while preserving high-level relationships between flagged used and unused dependencies. Hence, developers are provided with essential information but not overwhelmed.

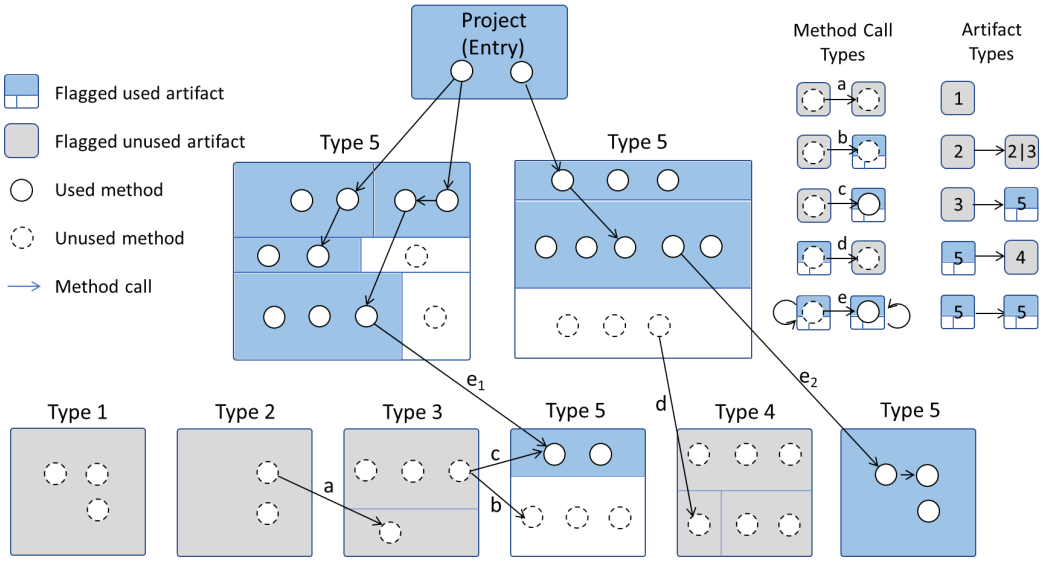


Figure 4.5: Classification of flagged used and unused dependencies.

The rule of classification follows the symbol of method call types and artifact types shown on the upper-right hand side. Method call types represent all the possible relationships between methods in two artifacts. Based on the incoming and outgoing method call types of an artifact, the artifact can be categorized into five types. Type 1 and Type 2 only have relationships with other flagged unused dependencies. Type 3 and Type 4 are more complicated and have relationships with flagged used dependency. Contrary to Types 1–4, we consider Type 5 as flagged used dependency and reachable from the entry code of the application.

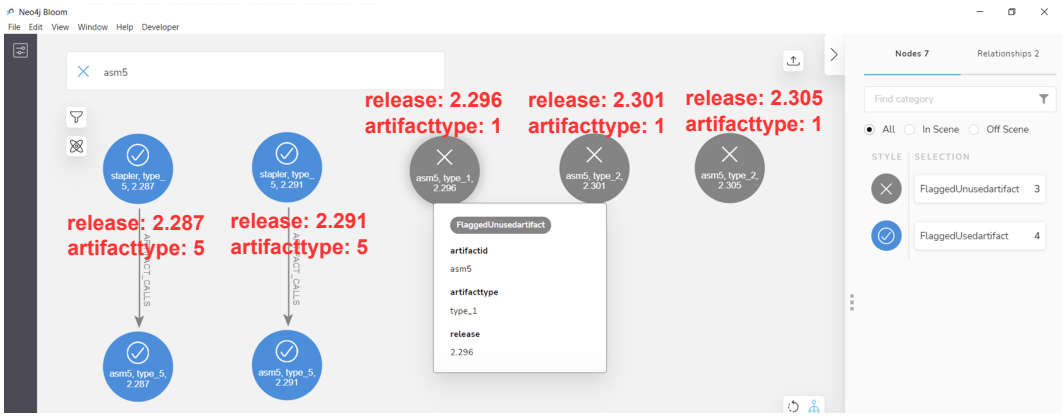


Figure 4.6: Neo4j Bloom for the visualization of the call graph in the artifact level after the classification of artifact types and the relationship between artifacts.

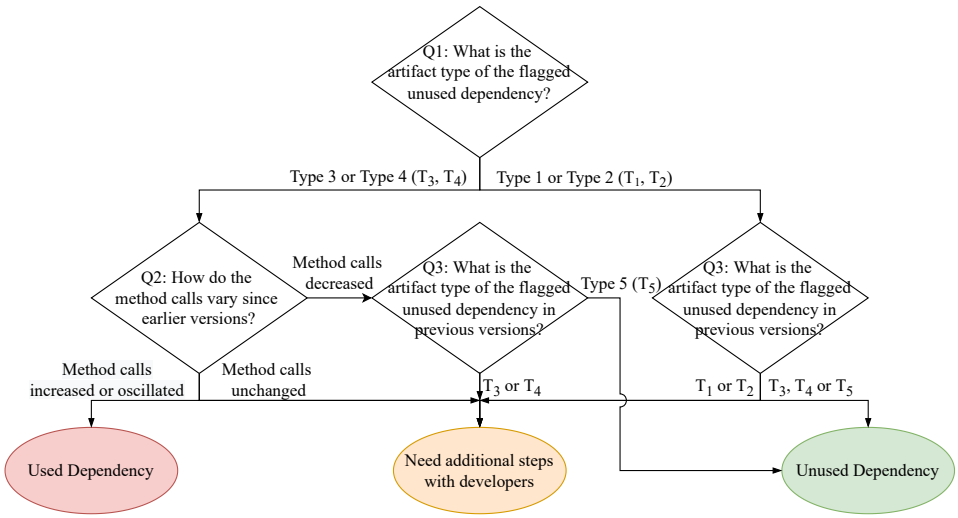


Figure 4.7: Decision process for the recommendation of individual flagged unused dependencies.

4.3 Analysis of the history of code changes

The high-level classification of a flagged dependency also helps us describe the code changes related to a dependency across versions. Analyzing and comparing to earlier versions might find crucial evidence that increase the confidence of flagged unused dependencies. For example, if part or all incoming method calls of a flagged dependency disappear after a certain version, this flagged dependency may no longer be needed. To convey the confidence of removing a flagged unused dependency based on the change of its usage, Fig. 4.7 provides a decision tree that depicts the decision process and its recommendations. Hence, developers may rely on these recommendations to prioritize which flagged unused dependency could be removed. Three questions are pinpointed as follows:

Q1 **What is the artifact type of the flagged unused dependency?** Every unused JAR artifact is categorized by the complexity of its relation with other artifacts. The flagged dependency that has more method calls across artifacts like type 3 and type 4 is handled differently in comparison to the flagged dependency with few methods calls across artifacts.

Q2 **How does the method call vary since previous versions?** If there is any method call removed since previous versions, this may be an indication of an unnecessary dependency. On the contrary, if the number of method calls increase compared to previous versions, the dependency should be retained. However, if the connection is unchanged throughout previous versions, it requires extra effort to distinguish the usage of the dependency. It is because the call graph tool is not sound, and possible to miss some features. To be safe, developers have to decide if it is required to be investigated further.

Q3 **What is the artifact type of the flagged unused dependency in previous versions?** This question compares both types 1,2 and types 3,4 to their artifact types in the earlier version. For example, if their artifact type in previous versions is type 5, it is obvious that some of the method calls have been removed since earlier versions. In this case, the flagged dependency is recommended as unused. On the other hand, if the artifact type is the same as in previous versions, the recommendation is needing additional steps with developers.

Generally speaking, the decision process is designed to be strict with giving recommendations to remove a flagged dependency. When there is evidence that method calls have reduced since an earlier version such as from type 5 to type 1–4, the flagged dependency is recommended as unused. On the other hand, when some method calls are observed to be increased or varied, the flagged dependency is considered as used (i.e. a false positive). For a flagged dependency that has no method calls changed in earlier versions, the flagged dependency needs additional steps with developers before taking further actions.

4.4 Design of the visualization for the dependency analysis

In section 4.2, Neo4j Bloom is introduced to visualize classified artifact types and their relationship to other artifacts. This section provides more details about the visualization for the dependency analysis by Neo4j Bloom. Since the dependency analysis has limitations and often requires developers to re-examine the correctness of the result, this work proposes a good visualization tool that may assist developers in decision-making and allow developers to efficiently understand how a dependency is used. By the design of this work, Neo4j Bloom is applied to visualize the relationships between flagged dependencies, so developers can compare the call graph at the artifact level in various releases. The workflow of analyzing dependencies with Neo4j Bloom is presented in a repository. The repository of the decision framework is public and can be found in `https://bitbucket.org/scam2022chingchichuang/static_dependency_analysis/`. The repository is composed of five main folders.

- The `RQ2` folder contains all the scripts to run the dependency analysis of all the specified releases.
- The `docker` folder provides a Dockerfile to replicate the software and its execution environment.
- The `neo4j-dump` includes the analysis result of three open-source projects that can be imported to Neo4j Desktop.
- The `depcleanfork` is a submodule that links to `opalAugmentedDepClean` branch of DepClean. In this branch, OPAL API is used to generate call graphs that augment the call graph generated by DepClean before the dependency analysis.
- The `fasten` is a submodule of an intelligent software package management system. This project provides an OPAL plugin for generating call graphs.

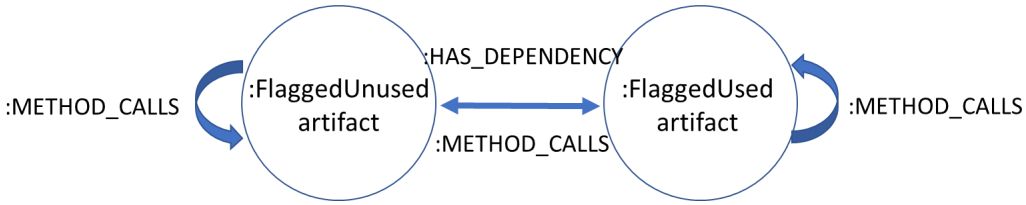


Figure 4.8: Neo4j definition

Table 4.1: The example of the dependency analysis by DepClean

Graph Database Model	Labels	Properties
Node	FlaggedUnusedartifact	artifactId, artifactReachability, artifactType, groupId, nodeId, owned, release, version
	FlaggedUsedartifact	
Relationship	METHOD_CALLS	weight, methodCallType
	HAS_DEPENDENCY	-

Once following the instructions of the repository, the result of an analyzed project is uploaded to the Neo4j graph database. The result can be visualized via Cypher Query Language [5] in Neo4j Bloom. In this work, three kinds of queries are respectively designed to assist developers to explore the dependency tree, the method calls to a dependency, and the call graph of a dependency in different releases.

4.4.1 Dependency tree

The design of a graph model for the dependency analysis is shown in Fig. 4.8, and the definition of properties is described in Table. 4.1. Each node has two kinds of relationships with other nodes. The first kind of relationship is called `HAS_DEPENDENCY`. This relationship is built according to the dependency tree declared in the POM file. A query example for a dependency tree is shown in List 4.1 which needs to be manually copied to Neo4j Bloom as in Fig 4.9. The result of the query is shown in Fig 4.10.

```
MATCH paths = (node1)-[:HAS_DEPENDENCY *0..1]->(node2)
WHERE node1.release='2.291' AND node2.release='2.291'
RETURN DISTINCT paths
// () rounded bracket represents a node.
// [] squared bracket represents a relationship.
// -> indicates the direction of a relationship.
// *0..X specifies the range of the length. Zero length is
   introduced to instruct Cypher to bind the last node.
// The word after a colon is a label.
```

Listing 4.1: Example of querying a dependency tree

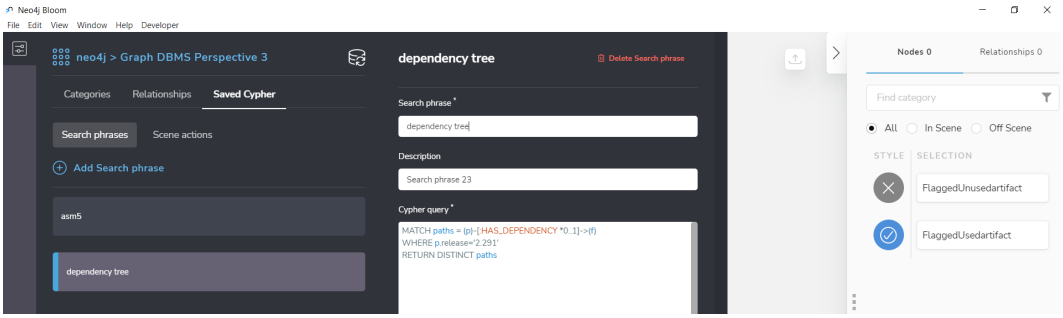


Figure 4.9: Cypher query in Neo4j Bloom

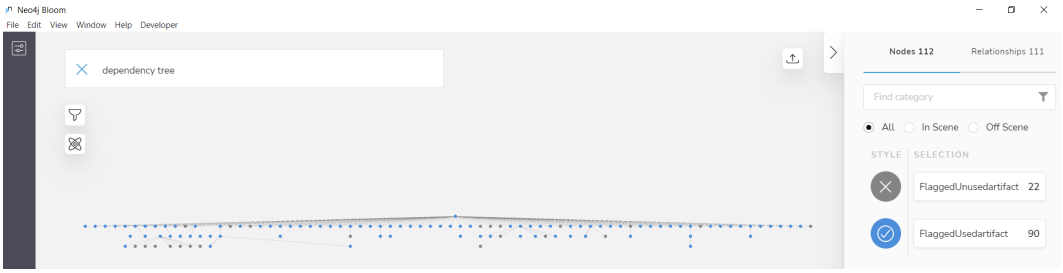


Figure 4.10: Dependency tree visualization of the Jenkins project in Neo4j Bloom which helps capture how a flagged dependency is declared in the POM file.

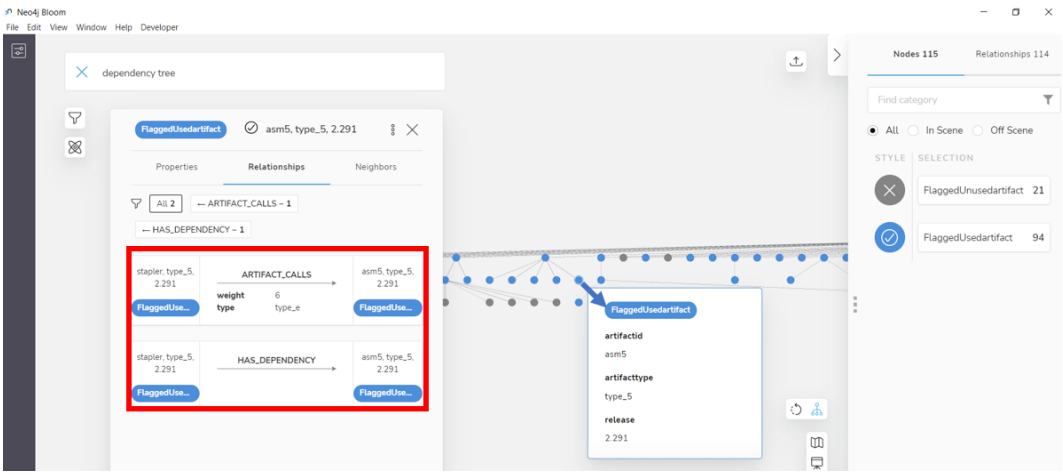


Figure 4.11: A pop-up window of Neo4j Bloom for relationships between flagged dependencies.

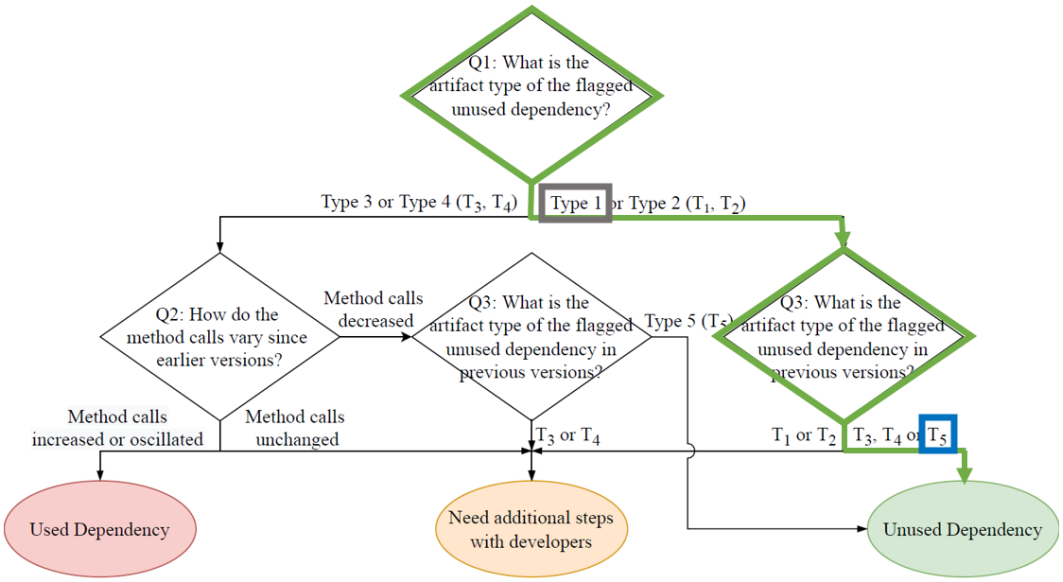


Figure 4.12: With the information of artifact types and the release, developers can follow the decision tree process to find recommendations of a flagged unused dependency.

4.4.2 Method calls of a flagged dependency

The second kind of relationship is called `METHOD_CALLS`. This relationship has two properties: `weight` and `methodCallType`. The `weight` indicates the number of its incoming and outgoing method calls while the `methodCallType` follows the definition in Fig 4.5. After displaying the dependency tree, the usage of a dependency can be found by double clicking the dependency icon such as `asm5` in Fig 4.11. After clicking, a pop-up window in Neo4j Bloom shows the `weight` of its incoming and outgoing method calls. Hence, developers may effectively see the usage of any dependency along with a dependency tree.

4.4.3 Decision process based on the call graph information

In addition to the dependency tree, call graphs of a dependency in different releases can be queried altogether in Neo4j Bloom. An example is provided in List 4.2. The result of this query has been shown earlier in Fig. 4.6. With the information of the artifact type and release obtained in Fig. 4.6, this work follows the decision tree and recommends that asm5 is unused dependency as shown in Fig 4.12.

```
MATCH paths1 = (n)-[:ARTIFACT_CALLS *0..1]->()
MATCH paths2 = ()-[:ARTIFACT_CALLS *0..1]->(n)
WHERE n.artifactid = 'zipkin-lens'
RETURN paths1, paths2
```

Listing 4.2: Example of querying a dependency tree

Chapter 5

Evaluation of the decision framework by ING project

After introducing how the decision framework is designed in the previous chapter, this chapter aims to evaluate the decision framework via the ING project. The chapter starts by stating the research question that the decision framework will experiment with. Next, the methodology used for the evaluation is explained. Lastly, the results of the evaluation are analyzed and used to discuss the research question of this chapter.

5.1 Research Question

***RQ1**: Can we systematically combine automated and manual analyses to improve the detection of unused dependencies of software projects at ING?*

Why: As shown in the motivating example of section 2.4, some of the detected unused dependencies in the enterprise user management application may be false positives. Reducing false positives by the automated decision framework can save the effort of developers in decision making, which encourages them to invest time in removing dependencies with high confidence and deals with less probable ones later. However, the recommendations of the decision framework have to be manually evaluated by developers, which may not be available for open-source projects. Hence, this work relies on developers at ING to judge and verify the correctness actively.

How: One enterprise user management application in the production is chosen because of its peculiar software structure which poses challenges for the state-of-the-art tool. After being presented with the suggestions of the decision framework, developers select some dependencies for testing based on their understanding of the usage of the dependencies. The result of the tests is compared to the recommendation of the decision framework.

5.2 Methodology

This work selected an enterprise user management application in the production for the evaluation because it is legacy software that certainly contains unused dependencies. The application is packaged as an EAR file which includes 144 Jar dependencies, and 43 of them are maintained by different groups at ING. The class and line of code of dependencies are 3050 and 211657 respectively. The project migrated from an inhouse-developed platform to Maven in 2019. Hence, there are 41 releases available to evaluate the decision framework.

To evaluate the recommendation, developers were guided through the results of the automated decision framework and were asked to provide their input: whether they agree with the recommendation and what is the main reason. Three developers who develop or maintain the project are invited. Two developers have worked on these projects for more than 4 years. One developer has maintained this project for almost 2 years. All of them have more than 5 years of working experience on Java projects.

Next, we remove the dependencies approved by developers from the project's POM file and execute the existing system and functionality tests. If the tests pass, we deploy our changes to the test environment of the software. This triggers a set of additional ING-specific checks that maintainers have to perform to validate the changes. Meanwhile, we also collect the system log to spot any unusual behavior – e.g., an error message. In the absence of any issue or concern from developers, we assume that the dependency can be successfully removed and our code changes can be merged to production.

This process is time-consuming and is taken very seriously by developers at ING. To use their time efficiently, we opt for removing multiple dependencies in the same merge request.

5.3 Result

Table 5.1: Dependency analysis result of the project at ING

Dependency Analysis Tool	Number of Flagged Dependencies	
	Used	Potentially Unused
DepClean	73	71[a]
DepClean + OPAL	85	59[b]
DepClean + OPAL + Decision process	98	46[c]

[a] From which 26 are direct, 45 transitive, and 0 inherited.
[b] From which 16 are direct, 43 transitive, and 0 inherited.
[c] From which 16 are direct, 30 transitive, and 0 inherited; we conclude that there are 10 `unused` and 36 `need additional steps with developers`.

The usage of dependencies is analyzed and presented in Table 5.1 which compares the results collected from different dependency analysis tools. The baseline is provided by DepClean which flags 73 used and 71 unused dependencies. After augmenting the DepClean

Table 5.2: Summary of the recommendations of the decision process and developers' feedback.

Recommendation	Developers' Decisions	Developers' Reasons	Number
Used dependencies (13)*	Agree with the recommendation	The functionalities of these dependencies are known and needed.	8
	Not sure about the recommendation	These are transitive dependencies and functionalities are unknown.	5
Need additional steps with developers (36)	Do not remove dependencies	The functionalities of these dependencies are known and needed.	8
		These are transitive dependencies and may become used in the future.	18
		The dependencies only contain javascript code.	4
	Can be removed	These dependencies have not changed for years.	6
Unused dependencies (10)	Do not remove dependencies	These are transitive dependencies and may become used in the future.	6
		These dependencies may be used in edge cases.	1
		The functionalities of these dependencies are known and needed.	2
	Can be removed	This is a duplicated dependency.	1

*In total, there were 98 flagged used artefacts. We selected the 13 artifacts that had been flagged as unused in the early stages of the analysis but were then discarded by our decision process.

call graph with critical OPAL edges, another 12 dependencies become accessible from the entry classes. Hence, the number of flagged used dependencies increases to 85 while the number of flagged unused dependencies decreases to 59.

Next, all the 59 flagged unused dependencies are classified as the corresponding artifact types and are compared to their classification in the earlier versions. By doing so, we find that the number of method calls in another 13 flagged unused dependencies has increased since earlier versions. Hence, these 13 flagged unused dependencies are recommended as ***used*** by the decision process. As a consequence, the number of potentially unused dependencies lowers again to 46. Within these 46 flagged unused dependencies, there are 10 flagged unused dependencies whose method calls reduce since an earlier version, so these 10 flagged unused dependencies are recommended accordingly as ***unused*** by the decision process. For the rest 36 flagged unused dependencies, their method calls are relatively stable, which calls for ***additional steps with developers***.

Table 5.2 summarizes the recommendations of the decision process and developers' feedback on 59 potentially unused dependencies flagged by DepClean+OPAL. For the recommendation as ***used dependencies***, developers agree with eight of them because they know the functionalities of these eight dependencies and are certain about their use cases. On the other hand, they are clueless about the other five recommendations since all these five dependencies are transitive and are barely noticed.

For the recommendation that ***need additional steps with developers***, the majority of them are declined by developers to remove due to several reasons. Firstly, some of their functionalities are known and needed by developers. Secondly, developers avoid excluding currently unused transitive dependencies in case they may become used after the dependency upgrade in the future. Thirdly, the dependency may aim to package other file formats such as javascript instead of adding java class files. In addition to the majority, developers decide that 6 dependencies can be removed since they have not used them for years. For the recommendation as ***unused dependencies***, 9 of them are declined by developers to remove because of the future upgrade of transitive dependencies, possible usages by edge cases, and the necessity of functionalities. Only 1 dependency in these recommendations is accepted by developers due to the existence of a duplicated dependency.

The result of the system and functionality test for dependency removal is presented in Table 5.3. Only 10 out of 59 unused dependencies flagged by DepClean+OPAL are forwarded to dependency removal tests. It is because many dependencies are declined by developers to remove due to safety concerns. For the recommendation of ***used dependencies***, three dependencies are selected for the tests, and all of them cause some failures of functionalities as expected. For the dependencies that ***need additional steps with developers***, six dependencies are chosen while half of them fail the tests. For the recommendation of ***unused dependencies***, one dependency is picked and passes the test.

5.4 Discussion

The combination of automated and manual analyses reduces false positives and helps developers prioritize the tests of unused dependencies. Results presented in Table 5.1

Table 5.3: System and functionality test for selected dependencies based on the recommendation and developers' feedback.

Recommendation	Developers' Reasons for Tests	Dependency Removing Tests
Used dependencies (3/13)⋆	Verify that dependencies are indeed used.	Test=3; Pass=0; Fail=3
Need additional steps with developers (6/36)⋆	Can be removed	Test=6; Pass=3; Fail=3
Unused dependencies (1/10)⋆	Can be removed	Test=1; Pass=1; Fail=0

⋆Only partial dependencies were selected for dependency-removing tests.

show that 12 dependencies that were initially flagged as unused by DepClean become used after we augment the call graph with critical OPAL edges. We conjecture three reasons that may explain this. First, the provided project heavily relies on dynamic proxies to invoke various implementations of services – this is the case for 10 out of 12 dependencies that implement such services. The dynamic proxy is one of the dynamic features in the Java language that is supported by OPAL. Hence, once we augment the call graph using OPAL, we effectively reduce false. Second, there are 2 out 12 dependencies that become used not because they implement dynamic features but because they are direct dependencies used by some of the previous 10 dependencies. Thirdly, it is noticed earlier that many of the frequent-occurred OPAL edges are overestimated by method implementations such as toString, hasNext, and toArray. However, none of these 12 dependencies become used due to overestimated method implementations since only critical edges are accepted. Hence, depending on the application, the degree to which our augmented analysis brings benefits will change. In particular, we anticipate that our augmentation brings more value to applications that rely heavily on dynamic features of Java, which is the case of the software project we study at ING.

The results also show the usefulness of the decision process. For the recommendation of ***used dependencies***, developers agreed that 8 out of 13 dependencies are used. For another 5 out of 13 dependencies that developers are uncertain about, they consider these dependencies as used because they are all transitive dependencies and the functionalities are unknown. When the functionalities of unused dependencies are unknown, removing them may cause potential errors in the application. For the dependencies that ***need additional steps with developers***, the majority of these dependencies are declined by developers to remove. This fact indicates that it is necessary for the dependency analysis tool to offer this kind of recommendation rather than merely providing a binary recommendation (used/unused). For the recommendation of ***unused dependencies***, developers are more concerned about considering them unused. Developers only accept this recommendation for the dependency that is duplicated.

To evaluate the recommendation and developers' feedback, ten dependencies are selected for system and functionality tests in Table 5.3. After developers remove 3 dependencies recommended as ***used***, each of them causes different failures which include error

messages in the system log and functionality breaks at the server. In other words, these dependencies are verified as actually used in the application. Hence, it shows that the decision process can indeed help us reduce false positives. For dependencies that ***need additional steps with developers***, the result shows that developers may not always be correct on the usage of the dependencies. Three out of 6 dependencies that developers claim to be unnecessary cause functional errors after they are removed. Hence, when the decision process recommends needing additional steps with developers, developers indeed need to take more efforts to investigate. For the only ***unused dependencies*** accepted by developers, it passes the test as expected. Therefore, with the help of the decision process, developers may prioritize how they plan to test and remove the unused dependencies.

However, the current design of the decision framework has a limitation and would falsely recommend a dependency as unused in some circumstances. For example, two dependencies are recommended as unused but are considered as needed by developers in Table 5.2. It is because we only select critical edges in the OPAL call graph. When we examine the OPAL call graph, we find that all target methods of these 2 dependencies occur more than once. Since we only select critical edges to augment the call graph, the edges created by these 2 dependencies are ignored. Another corner case that is not covered by our approach is when transitive dependencies are directly called by the application. However, we do not observe this corner case in our data.

5.5 Threats to validity

5.5.1 Construct

The approach to augmenting the call graph is designed according to the context of the provided ING project. Since the provided web application is developed a decade ago, they use the feature of the dynamic proxy to conveniently invoke various services before the technique of the dependency injection becomes popular. For the project developed in recent years, the context of the software development must have changed and our approach should be adjusted to fit the different context. Specifically, the call graph construction tool, the mechanism of selecting critical edges, and the questions of the decision process may need to be adapted for the targeted project.

5.5.2 Internal

The system and functionality test rely on the experience of developers and the identification of error messages in the system log. However, even the senior developers may not know all the details in the dependencies maintained internally at ING. Also, the time allocated for the tests is limited, so some dependencies have to be tested within a batch. Although the result is expected to be the same as being tested individually, this premise has not been verified yet. Moreover, it is assumed that the error caused by the removed dependency can be triggered in a short time, which may not always be the case. Some faults may exist in the system for a long time without causing error messages.

Chapter 6

Evaluation of the decision framework by open-source projects

6.1 Research Question

***RQ2**: Is our decision framework to detect unused dependencies confirmed by the commit history of other open source projects?*

Why: Since the selected enterprise user management application at ING is maintained by the production team, the available testing windows and the capacity of developers are limited. Hence, this work further resorts to open-source projects as alternatives to evaluate the decision framework. In the open-source projects, dependencies that were previously deleted are valuable and can be considered as ground truth to evaluate recommendations by the decision framework.

How: This work inspects the history of three open-source projects: Jenkins, Zipkin, and Onedev. Projects are selected based on the number of declared dependencies, releases, and commits on removing dependencies. For each project, this work collects all the commits that add or remove dependencies and compare the reasons behind these changes with the proposals from our decision framework.

6.2 Methodology

Among three open-source projects, Jenkins is used in previous work [37] to evaluate DepClean while Zipkin and Onedev are selected from a set of 50 projects collected with the search tool *SEART*[1]. With SEART, this work retrieves 50 top Java projects available on Github that have more than 5k stars, more than 50 releases, and was active in the first quarter of 2022. The result of the top 50 projects by the project size can be found in this link. These projects are then filtered according to the following criteria: 1) the project uses Maven to manage dependencies (e.g., Gradle projects are discarded), 2) the project has a

[1] https://seart-ghs.si.usi.ch/

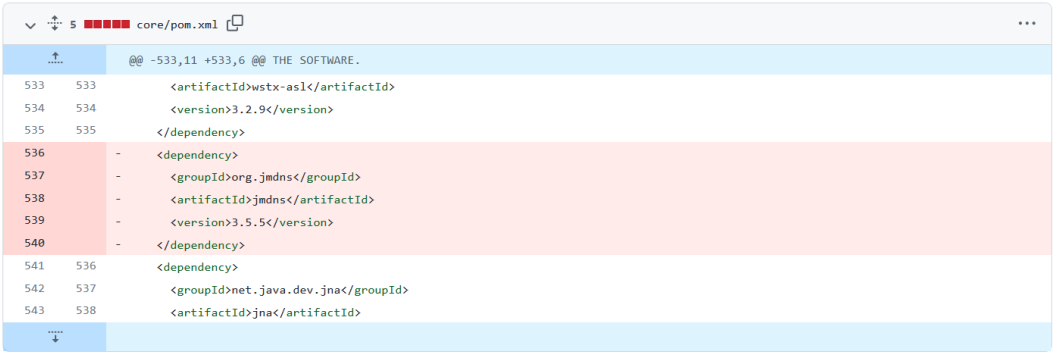


Figure 6.1: An example of removing a dependency in Jenkins core's POM file.

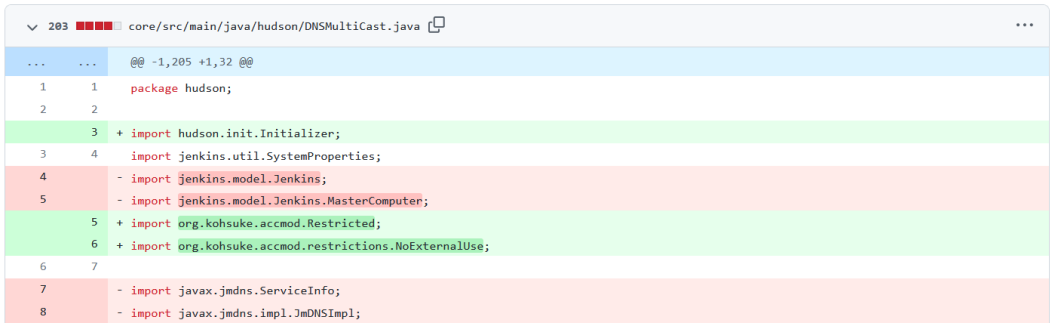


Figure 6.2: An example of removing source code related to the removed dependency.

single main module that can be used for analysis (multi-module analysis is not supported yet), and 3) the project has at least 5 commits that remove dependencies.

Next step is to collect all the commits that remove or add dependencies in every release of the selected projects. For the removed dependencies, this work examines the code changes and commit messages to assess why they are removed. For the added dependencies, this work checks whether that dependency is brought back in later releases. Any commit that does not reveal the reason for removing a dependency is discarded. With this manner, it ensures that the collected commits are valid ground truth. The spreadsheet of ground truth can be found in this link.

This work then determines the reasons behind removing dependencies by reading the commit messages and inspecting code differences. This work then divides commits into three groups (R_1) replace dependency, (R_2) remove code and dependency, and (R_3) only remove dependency. Reasons R_1 and R_2 imply that the dependency is still needed by the project and some other reason lies in its removal from the dependency specification (e.g., security issues, API migration, etc.). Reason R_3 clearly indicates that the dependency is unused – hence, this work compares these cases with the recommendations provided by the framework (cf. Fig. 4.7).

For example, Fig 6.1 and Fig 6.2 is one of collected commits in Jenkins project. This commit can be found in this link. A dependency called jmdns is removed from Jenkins

core's POM file in this commit. Moreover, some pieces of code related to jmdns are also removed together in this commit. This reveals that the dependency is still used before being removed. Hence, this commit is categorized in the group of R_2 (remove code and dependency).

6.3 Result

The results of the consistency between the reason for removing a dependency in the open-source projects and the recommendation of the decision process are presented in Table 6.1. This work divides each dependency removal by the different reasons behind that change: R_1, R_2, R_3 (as explained in Section 6.2).

For each reason for removing dependencies, the dependency usage immediately before being removed can be inferred as ground truth to compare the recommendations of our proposed decision process and DepClean. For any recommendation that is inconsistent with the reasons for being removed, the result is labeled with an asterisk symbol in Table 6.1.

In sum, the results show that only 2 recommendations from our decision process are inconsistent with the actual reasons for being removed. This result is far better than DepClean's recommendation, where 21 recommendations are not reflected in the commit history. In addition, there are 23 dependencies that our decision process can not determine their usage. In those cases, we need additional steps with developers.

6.4 Discussion

The recommendations of the decision process are consistent with the history and are cautious not to remove false positives. The results show different fingerprints of three open-source projects that can help us check if the recommendations of the decision process are consistent with the history.

For the recommendation of ***used dependencies*** by the decision process, 28 of 30 recommendations are correct since R_1, R_2 indicate that those dependencies are still used or maintained. Specifically, if a dependency is replaced by another dependency or removed along with some source code, it means that the removed dependency is still used before they are removed. Hence, the recommendation of not removing them is correct. On the other hand, 2 of 30 recommendations contradict the developers' reasons for removing dependencies. However, when the detail of the commit history is investigated, it is found that these dependencies are removed either because they are duplicated[2] or because they become provided[6], meaning that they are still used before they are removed. Thus, the suggestion of the decision process matches all of the commit histories in this category. For the recommendation of ***unused dependencies*** by the decision process, all of them match the history of 2 commits in the Jenkins project that only remove unused dependencies.

For the recommendation that ***needs additional steps with developers*** by the decision process, 11 out of 16 dependencies are used according to the ground truth of commit history. However, these 11 dependencies are unreachable from the entry classes, which means there are some missing method calls to these 11 dependencies in the call graph. In this

Table 6.1: Evaluation of the decision process by checking the reasons of removing dependencies in the open-source projects.

Project Name	Reasons for Removing Dependencies	Dependency Usage Immediately Before Being Removed (GT)		Recommendations by the Decision Process				Recommendations by DepClean	
		Used (46)	Unused (9)	Used (37)	Unused (2)	Need additional steps with developers (16)		Used (29)	Unused (26)
						$T_1 \| \| T_2$	$T_3 \| \| T_4$		
Jenkins core (24)	Replace dependency(R_1)	4	0	3	0	0	1	3	1*
	Remove code and dependency(R_2)	12	0	12	0	0	0	12	0
	Only remove dependency(R_3)	0	8	2*	2	2	2	2*	6
Zipkin server (23)	Replace dependency(R_1)	2	0	2	0	0	0	1	1*
	Remove code and dependency(R_2)	20	0	14	0	2	4	7	13*
	Only remove dependency(R_3)	0	1	0	0	0	1	0	1
Onedev server (8)	Replace dependency(R_1)	1	0	1	0	0	0	1	0
	Remove code and dependency(R_2)	7	0	3	0	2	2	3	4*
	Only remove dependency(R_3)	0	0	0	0	0	0	0	0

*The recommendations contradict to the commit history (GT).

GT: *ground truth*, $T_1||T_2$: *Artifact type1 or type2*, $T_3||T_4$: *Artifact type3 or type4*

circumstance, DepClean flags these 11 dependencies as unused. In contrast, our decision process does not falsely classify them as unused because artifact types in previous releases are also considered. Since the decision framework does not find evidence to support whether the dependency is used or unused, the decision process is cautious and recommends taking additional steps with developers.

Likewise, when DepClean is used to decide the usage of these dependencies, the analysis result shows that 26 dependencies in all three projects are considered unused; however, nineteen of these recommendations contradict the ground truth inferred from the commit history. In contrast, since our decision process considers the history changes of method calls and artifact types, some of these 19 dependencies are recommended as used by the decision process while others need additional steps with developers. In this manner, even though the call graph can not capture some dynamic features in the Java language, the decision process does not wrongly suggest developers remove false positives. This feature is crucial for the production environment.

6.5 Threats to validity

6.5.1 External

Although the history of open source projects is applied to confirm the recommendation of the decision framework, the system and functionality test can not be executed in the open-source projects as being done in the project at ING. In addition, only a few open-source projects regularly remove the unnecessary dependencies like Jenkins. For most of the studied open-source projects, developers remove dependencies usually when the dependency is replaced or the software updates. This fact presents a difficulty for us to gain more data to support our decision process.

Chapter 7

Conclusions and future work

In this work, a decision framework to reduce false positives of unused dependency detection is proposed. The decision framework extends the state-of-the-art dependency analysis tool DepClean and analyzes one industrial Maven project at ING along with three open-source projects.

For the project at ING, it is found that the augmented call graph helps reduce 12 false positives out of 71 unused dependencies detected. Also, the decision process based on the classification of the relationship between dependencies helps reduce 13 false positives. Hence, a decision framework of these two approaches filters out one-third of false positives of unused dependencies. The decision process further categorizes the remaining two-third unused dependencies according to their release history, which allows developers to decide which dependency could be removed with ease or not.

Furthermore, the recommendations of the decision process are verified to be consistent with the reasons for removing dependencies in three selected open-source projects. Even though the dynamic feature of Java hampers the accuracy of the dependency analysis tool and creates false positives, the decision process relies on the changes in the relationship between dependencies and successfully points out 11 dependencies that could have become false positives.

In future work, our decision framework can be extended in different ways: improve the precision of OPAL when building a call graph for large software projects; analyze hierarchical multi-module Maven projects and see how the decision process needs to be adjusted; expand our methodology with other call graph tools to enhance the soundness to a great extent. Furthermore, it would be interesting to expand the study to understand how the visualization tool helps developers understand why dependencies are classified as unused.

Bibliography

[1] Transformation of Java Bytecode to KDM Models as a Foundation for Dependency Analysis. https://oceanrep.geomar.de/id/eprint/16434/. Accessed: 2022-07-01.

[2] Commit history of removing a duplicated dependency. https://github.com/jenkinsci/jenkins/commit/b37dc3242475f1ee5f605deec2954a6b8b07bb64.

[3] ING. https://www.ing.nl/particulier/index.html. Accessed: 2022-07-01.

[4] Neo4j Bloom. https://neo4j.com/product/bloom/, . Accessed: 2022-08-24.

[5] Cypher Query Language. https://neo4j.com/developer/cypher/, . Accessed: 2022-08-24.

[6] Commit history of removing a provided dependency. https://github.com/jenkinsci/jenkins/commit/63c7c2f70392ad2a8d6aed83d6593e27995017e5.

[7] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

[8] Gabriele Bavota. Using structural and semantic information to support software refactoring. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1479–1482. IEEE, 2012.

[9] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701. IEEE, 2013.

[10] Paolo Boldi and Georgios Gousios. Fine-grained network analysis for modern software ecosystems. *ACM Transactions on Internet Technology (TOIT)*, 21(1):1–14, 2020.

[11] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.

[12] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[13] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 2020.

[14] Michael Eichberg and Ben Hermann. A software product line for static analyses: the opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[15] Norman E Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370, 2000.

[16] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 209–220, 2018.

[17] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. A survey on software coupling relations and tools. *Information and Software Technology*, 107: 159–178, 2019.

[18] Rani Geetika and Paramvir Singh. Dynamic coupling metrics for object oriented software systems: a survey. *ACM SIGSOFT Software Engineering Notes*, 39(2):1–8, 2014.

[19] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. Is static analysis able to identify unnecessary source code? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(1):1–23, 2020.

[20] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. Api beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client–api usages. *Journal of Systems and Software*, 184:111134, 2022.

[21] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software*, 183:111097, 2022.

[22] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 184–196, 2020.

[23] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249. IEEE, 2019.

[24] Usman Ismail. Incremental call graph construction for the eclipse ide. *University of Waterloo Technical Report*, 2009.

[25] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[26] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):1–50, 2019.

[27] Konner Macias, Mihir Mathur, Bobby R Bruce, Tianyi Zhang, and Miryung Kim. Webjshrink: a web service for debloating java bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1665–1669, 2020.

[28] Steven Raemaekers, Arie Van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 221–224. IEEE, 2013.

[29] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 43–48, 2017.

[30] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–261, 2019.

[31] Simone Romano and Giuseppe Scanniello. Exploring the use of rapid type analysis for detecting the dead method smell in java code. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 167–174. IEEE, 2018.

[32] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[33] Giuseppe Scanniello. An investigation of object-oriented and code-size metrics as dead code predictors. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 392–397. IEEE, 2014.

[34] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[35] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. Trace-based debloat for java bytecode. *arXiv preprint arXiv:2008.08401*, 2020.

[36] César Soto-Valero, Thomas Durieux, and Benoit Baudry. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1021–1031, 2021.

[37] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):1–44, 2021.

[38] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation. In *Asian Symposium on Programming Languages and Systems*, pages 69–88. Springer, 2018.

[39] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1049–1060. IEEE, 2020.

[40] Núria Bruch Tàrrega, Miroslav Zivkovic, and Ana Oprescu. Measuring the impact of library dependency on maintenance. In *SATToSE*, 2020.

[41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[42] Junjie Wang, Song Wang, and Qing Wang. Is there a" golden" feature set for static warning identification? an experimental evaluation. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

Appendix A

Glossary

In this appendix an overview of frequently used terms and abbreviations are given.

artifact type: The classification of artifacts according to their relationships (method calls) with other artifacts.

augmented call graph (CG): The combination of Depclean's call graph and OPAL's critical edges.

call graph: The call graph represents the calling relationships between methods, classes or artifacts.

critical edge: After building a call graph using OPAL at the method level, edges with target methods that have unique implementations are called critical edges.

decision framework: The decision framework is composed of three parts as shown in Fig 4.1.

decision process: The decision process is the last stage of the decision framework.

DepClean: DepClean is a state-of-the-art dependency analysis tool.

dependency analysis: The dependency analysis is a process of finding the usage of declared dependencies.

dependency tree: The dependency tree describes how dependencies are declared.

dynamic dependency analysis: The dynamic dependency analysis collects runtime information and infers the usage of declared dependencies.

dynamic feature: The features that allow the program to change at runtime.

dynamic proxy: The dynamic proxy is one of dynamic features. It allows developers to extend or modify existing functionalities and choose to invoke any of them at runtime.

enterprise user management application: Enterprise user management application is an enterprise web application.

entry class: The entry class is a starting point of a call graph when collecting all the possible routes.

flagged dependency: The flagged dependency is the output of the first stage in Fig 4.1.

Maven: Maven is a software package management tool. This work only chooses projects that are managed by Maven.

method call type: Method call type is the classification of method calls between artifacts.

Neo4j Bloom: Neo4j Bloom is a free visualization tool for Neo4j graph database.

OPAL: OPAL is a static analysis platform for Java bytecode analysis such as call graph.

POM file: A Project Object Model or POM file is where developers declare the usage of dependencies with MAven.

precision: Precision is to evaluate what is the percentage of method calls in a call graph that are actually invoked at run time.

soundness: Soundness is defined to evaluate what is the percentage of method calls invoked at run time can be found in the call graph.

static dependency analysis: The static dependency analysis analyzes bytecode offline and explore the usage of declared dependencies.