2021 2nd Global Conference for Advancement in Technology (GCAT)
Bangalore, India. Oct 1-3, 2021

JarBot: Automated Java Libraries Suggestion in JAR Archives Format for a given Software Architecture

P. Pirapuraj
Department of Information & Communication Technology, Faculty of Technology, South Eastern University of Sri Lanka,
Oluvil, #32360, Sri Lanka
pirapuraj@seu.ac.lk

Indika Perera
Department of Computer Science and Engineering, Faculty of Engineering University of Moratuwa,
Sri Lanka
indika@cse.mrt.ac.lk

***Abstract*— Software reuse gives the meaning for rapid software development and the quality of the software. Most of the Java components/libraries open-source are available only in Java Archive (JAR) file format. When a software design enters the development process, the developer needs to manually select necessary JAR files via analyzing the given software architecture and related JAR files. This paper proposes an automated approach, JarBot, to suggest all the necessary JAR files for given software architecture in the development process. All related JAR files will be downloaded from the internet based on the extracted information from the given software architecture (class diagram). Class names, method names, and attribute names will be extracted from the downloaded JAR files and matched with the information extracted from the given software architecture to identify the most relevant JAR files. For the result and evaluation of the proposed system, 05 software design was developed for 05 well-completed software project from GitHub. The proposed system suggested more than 95% of the JAR files among expected JAR files for the given 05 software design. The result indicated that the proposed system is suggesting almost all the necessary JAR files.**

Keywords*— *Java Archive (JAR); software architecture; class diagram; code reuse; bytecode analyzing: WordNet; N-gram technique.

I. INTRODUCTION

Developers use external libraries to speed up the development and decrease Software Project Manufacturing Costs. It is not an easy process to rightly use libraries from third parties [1]. When developing software applications, software developers depend on frameworks and public application program interfaces. When programming with an API, customers either have to use existing documents or codes to direct them using the target API [2].

The programmer should have various levels of knowledge to effectively use certain libraries and sample code, from the name of a class providing certain functionality to various methods calling upon multiple objects performing a particular task [3]. If the developer is not an expert in the domain of a particular project and he/she is fresh to the software development process, including libraries into the project will be a very big challenge. According to the current state of software companies, when the software design enters into the development process, few software libraries need to be included in starting the development process. If the company has an expert in the field/ context of the software, he/she will suggest few appropriate libraries. Otherwise, the developer needs to download the relevant libraries set and check the suitability of the libraries with the project to select relevant libraries. It will be a time-consuming task.

Indeed, in many languages, Java has a unique benefit: Without recompilation, the Java program operates on virtually any known software and hardware architecture. Java encourages reuse by componentization and container classes JAR (pre-compiled versions of the components) [4].

When configuring the Java development environment (installing JDK and JRE), some Java prewritten libraries will be included by default in JAR format inside the JDK folder. All Integrated development environments, such as Eclipse and NetBeans, include all the libraries from JDK when developing Java projects. All those JAR files included inside the JDK can be used publically in all the projects.

Most of the third-party libraries which are in JAR archive file format are not available inside JDK. But, those libraries are available on some websites as free. Selecting suitable libraries from those websites is challenging unless the project developer is an expert in the project context.

On the other hand, if all the necessary JAR files for a project are known before the implementation process, one of the software project management and comprehension tool like "Apache Maven" can be used via mentioning all the necessary JAR files inside the "Dependency" tag of POM.XML file in the project. All the dependency fill will be automatically added when executing the project using the project management and comprehension tool. If the developer does not know about the necessary JAR files before the implementation process, using the tool mentioned above is meaningless.

This paper proposes a framework to suggest JAR files automatically based on the information extracted from the given software architecture (Class diagram). The overall idea is to extract information from given software architecture (Class diagram), search and download all the related JAR files from the internet based on the extracted information, and analyze the suitability of all the downloaded JAR files with the given software architecture, select the most relevant JAR files among them via rating them in the analyzing process, and finally, suggest the selected JAR files as suitable libraries for the given software architecture.

978-0-7381-3215-0/21/$31.00 ©2021 IEEE

The remainder of the paper is organized as follows. Section II presents the Literature review of this study, Section III describes about the proposed system, giving an overall picture, describing how information is collected from software architecture and how it can be used to download JAR files, how to analyze those downloaded JARs and how to select most relevant JAR files. Section IV reports the result and evaluation. After a discussion in Section V, Section VI concludes.

II. Related work

There is no fully related research to this study, but there are few relevant works. This work included some techniques used in our earlier work [5]. Erik Linstead and et al. said that automated analysis is essential to drastically growing software repositories to understand software structure, function, complexity, and evolution [6].

Selene is a Code Recommendation System to suggest code while typing anything in IDE (ECLIPSE) from code repositories based on typed text in IDE. The most relevant code is selected by giving a local similarity [3]. Anh Tuan Nguyen and et-al. said that the present projects use Application Programming Interfaces (APIs) widely: even the "HelloWorld" program invokes an API strategy. Their research has proposed a tool APIREC, based on the programming code changes, and the proposed tool suggests the most relevant API calls. They have used the n-gram technique and some other machine learning techniques [7].

Another study performed by Santiago Vargas-Baldrich and others [8] on bytecode analysis and dependencies of open-source tags or categories defining features like application domains, programming languages, operating systems, etc. in the fields of browsing improving, looking for, and finding processes in large repositories. They developed a novel approach called SALLY to automatic tagging of closed-source (only bytecode is available) projects.

A. T. Nguyen and T. N. Nguyen have suggested two novel approaches and tools, such as GraLan and ASTLan, for API recommendation. GraLan, a graph-based statistical language model, suggests API based on the calculation of appearance probabilities of source code corpus. They build an API suggestion engine using GraLan and ASTLan supports the suggestion of common syntactic templates. In this research, the said that n-gram statistical language model faces challenges in catching the patterns at more elevated levels of abstraction because of the crisscross between the sequence nature in n-grams and the structure idea of syntax and semantics in source code [9]. Through their suggested API suggestion engine, they overcome the challenges mentioned above. Their approach deals with the program inside the IDE, but we search libraries from the internet, in that context, we need the n-gram technique.

In another research, the same problem was handled by Xiaoyu Liu and et al. [10], but they start from the result for the API recommendation. At the same time, API calls of top-10 API candidates were identified by GraLan [9], but they did not rely on code change history. They eliminate the weakness of GraLan by employing a discriminative re-ranker. In this way, they suggested a tool called RecRank. This novel discriminative positioning methodology utilizes a novel sort of highlights dependent on using ways to naturally suggest top-1 APIs dependent on the top-10 API applicants recommended by GraLan.

Another study conducted by Hussein Alrubaye and et-al. About third-party library migration. There was an urgent need to support developers in migrating their third-party libraries, and they have developed a tool called MigrationMiner. The tool using Abstract Syntax Tree (AST) code representation to migrate between two third-party libraries. They give GitHub open source projects as input, and the MigrationMiner extracts the following information from software projects as commit ID, commit date, developer name, and commit description. Using the information mentioned above, the MigrationMiner detects migrations between third-party Java libraries [11].

NonDex is a tool for detecting and debugging wrong assumptions on Java APIs. This means, sometimes client code can fail by applying an underdetermined API. NonDex helps to detect and debug such fails proactively. The tool was designed to detect the wrong assumptions by analyzing the behavior in the execution time [12].

Massimiliano Di Penta and et al. [4] proposed an automatize approach to identify the license of JAR achieves combining code-search engine use with the automated classification of licenses found in the JAR's text files. For this task, they used information decompiled from the bytecode of its classes to query a code search engine, such as class name and package name.

For most of the bytecode analyzing project, the ASM Bytecode Manipulation Framework is used to obtain class names, class fields, method names, and method arguments from bytecode [13] And Apache Lucene [14] is used to split the extracted identifiers from bytecode by camel case and stemmed.

III. proposed System and Methodology

The proposed system JarBot is the modified version of our earlier work[5] which suggests relevant source code files and source code snippets from source code forges (GitHub, SourceForge, etc.) when a software architecture enters into the development process. But, most of the Java libraries are in JAR archive file format, our earlier developed framework will not suggest the necessary JAR files. Therefore, the proposed system in this paper includes a few more fresh pieces of software components to handle JAR achieve files.

A. Proposed System

The JarBot includes several pieces of software components. The system starts with software architecture (Class diagram) in XML format to extract a few information (class name, methods name, and attributes name) to do the JAR files suggesting process. The following are the proposed system's major processes, I. Extracting important information from the given XML file (Class diagram), II. Crawl some JAR files from the internet based on the information extracted from the given software architecture, III. Extracting information from the downloaded JAR files, IV. Identifying the most relevant JAR files via comparing both information extracted from the given software architecture and downloaded JAR files.

from the given XML file (Class diagram), II. Crawl some JAR files from the internet based on the information extracted from the given software architecture, III. Extracting

information from the downloaded JAR files, IV. Identifying the most relevant JAR files via comparing both information extracted from the given software architecture and downloaded JAR files.

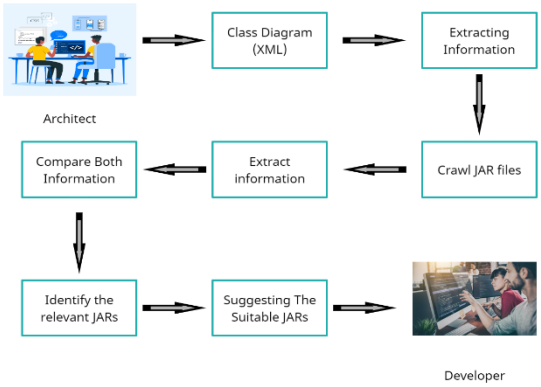

Fig 1. Data flow diagram of the proposed system

Figure 1 describes the data flow diagram of our proposed system. It starts with software architecture (class diagram) in XML format, extracts information from the architecture, crawl few JAR files based on the information extracted from the architecture, extract data from downloaded JAR files, compare the both information extracted from given software architecture and downloaded JAR files to identify the most relevant JAR files, and finally, suggest the set of suitable JAR files to the given software architecture.

B. *Methodology*

The JarBot starts with an XML file (class diagram), and information from the file is the input and starting point of our proposed system. The following module is the essential part of the proposed system.

XMLExtractor – For this module, we used the module created in our earlier work, the detailed information about the XMLextractor available in our earlier work [5]. The javax.xml package was used to implement this module, which has two main classes: *DocBuilder*: Define the API for obtaining DOM instances from the XML document *DOCBUILDERFactory*: define the API for the creation of the DOM objects tree from the XML documents. The purpose of this module is to extract information from the software architecture.

Java2s and jar-download Crawlers – Two specialized crawlers were created for downloading relevant JAR files. There are so many websites for downloading JAR files, but java2s and jar-download are two websites that make the downloading process easy. The JSOUP library is used to develop the crawlers mentioned above. Normally, the crawler starts with a keyword and a seed URL, the information extracted from the given software architecture are the keywords, and the URLs of the websites mentioned above are the seed URLs.

JARExtractor – This module aims to unpack the downloaded JAR files and extract information from them. All the JAR files have "META-INF" folder in them. Some JAR files are having ".txt" files while some are having ".MF" files inside the "META-INF" folder. The JARExtractor module collects some information from those ".txt" and ".MF" files.

ASMExtractor – Along with the information collected using JAR Extractor, a few more information (class names, class fields, method names, and method arguments) need to be collected from all the class files (.class bytecode files) are included inside all the JAR files. ASM Bytecode Manipulation Framework is used to implement this module to accomplish the task mentioned above.

ApacheLuceneSplitter – Most of the identifiers (class names, class fields, method names, and method arguments) obtained from all the downloaded JAR files using ASMExtracter were connected words (e.g. ListView(), lstCon, and writeTag()). This module implemented using Apache Lucene to split the identifier, which are connected words.

WordIdentifier – The results of the ApacheLuceneSplitter module were few real dictionary words and some meaningless abbreviated identifiers. The WordIdentifier module aims to identify the real meaningful words from the meaningless abbreviated identifiers obtained by the ApacheLuceneSplitter. If this module cannot identify the abbreviated term, the N-gramChunker module is used, and again this module will be used to determine the word. Stanford Spellchecker was used to implement this module.

N-gramChunker – As discussed earlier, the WordIdentifier identifies the real meaningful word from the abbreviated term. Sometimes, the WordIdentifier cannot identify the meaningful words from the abbreviated term. This module aims to make chunks from the abbreviated identifiers when the WordIdentifier cannot identify the real words. N-gram is an NLP technique, depends on the value of N and divides a word into chunks. For example the word is "rect" and N=2, the chunks are "re", "ec", "ct".

WordNet – WordNet is a lexical database of an English word and sense relations. A sense is a particular meaning of a word. WordNet provides the synset for each sense of a particular word, a list of synonyms for the sense. This module aims to identify the synonym of all the words identified from the above process. The JAWS Java library is used to develop this module. This module is used in two places in the proposed system: searching JAR files from the internet and finding a synonym of the words identified from the downloaded JAR files in the comparison process.

Analyzer – A large number of words will be produced from the modules mentioned above. The final task is to analyze the word pool extracted from the download JAR files and words extracted from the given software architecture to identify the most relevant JAR files from the downloaded JAR files. We used an equation derived in our earlier work [5] ($M = Mi + 100 / N$ where M is denoted marks are to be given for JAR file, Mi is denoted initial marks for each iteration, and N is denoted the number of identifier extracted from the XML file(class diagram)) for assign marks for matching words from the both way (words from the software architecture and words from the downloaded JAR files).

IV. Result and Performance Evaluation

In this section, the set of experiments conducted on the proposed system to validate it, evaluate its performance, and the datasets used to evaluate it are described in detail.

A. Input

As described in section II, the proposed system starts with software architecture in XML formats. Five well-completed software system from the GitHub have been selected, and the software design (class diagram) were drawn for the selected five software system and export it as an XML file. The details of the necessary JAR files were collected from the POM.xml files of the targeted projects from GitHub (because all the targeted projects are maven projects, and all the necessary JAR files are indicated inside the dependency part of the POM.xml file).

Table I includes the following: all the targeted projects from GitHub, the sample of the included JAR files, and the sample of the included class files of each project. Table II shows all the included JAR files, class files, and methods inside the targeted 05 projects. The "fastjson" was a bigger project among the 05 projects, including 189 classes, 721 methods, and 62 JAR files used to implement the project. The second-largest project was "Minim" which included 125 classes, 413 methods, and 11 JAR files were used to implement the project.

TABLE I. : SELECTED PROJECTS AND THE SAMPLE OF INCLUDED JAR FILES AND CLASSES

Projects Name	Sample of included classes	Sample of included JARs
fastjson	JSONPatch, AnnotationSerializer, ClassWriter	plexus-compiler-javac, javax.servlet-api, retrofit
AdyenPayments	Credentials, Payment, RetrieveRecurringCard Details, Credentials	adyen-axis-ws-client, commons-codec, wsdl4j
JFeatureLib	ThreadWrapper, LaplaceFilter, FuzzyOpponentHistogram	Imageanalysis, lire, args4j, commons-io
Minim	AudioListener, AudioRecordingStream, AudioOut	Jl, tritonus-share, mp3spi
soundcloud	Playlist, SoundCloud, Track	com.soundcloud.api, gson, httpclient

The class names and the method names of the targeted 05 projects were used to draw the class diagram for the targeted 05 projects, after drawing the class diagrams of all the projects, converted into an XML file. The XML files were the input for the proposed system. The expected output was the 101 number of JAR files.

The drawn class diagrams were given to the proposed system as input in XML file format. The XMLextractor module of the proposed system was used to extract information (class names and method names) from the given XML files. Most of the extracted information (identifiers) were connected words. The ApacheLuceneSplitter module was used to split the identifier, which is connected words. The extracted and split word pool were the keywords for the crawler modules.

B. Result of Crawlers

As we described in section II, the JarBot has two types of crawler, which are *Java2s* and *jar-download* for the two selected websites. The inputs for the crawlers are the information extracted from the given software architectures (class diagrams), and the seed URLs were the URL of the two websites mentioned above. Both the crawlers fetched 423 JAR files all together for the given information from both targeted web site. There were 97 expected JAR files in the downloaded JAR files pool out of the expected 101 JAR files.

TABLE II. : DETAILS OF THE JAR FILES, CLASSES, AND METHODS ARE INCLUDED IN ALL THE TARGETED PROJECTS FROM GITHUB

Projects Name	Number of JAR files	Number of Classes	Number of Methods
fastjson	63	189	721
Adyen	08	08	13
JFeatureLib	13	77	264
Minim	11	125	413
soundcloud	06	06	192

C. Result of other modules of the proposed system

After the crawling process, the rest of the proposed system modules start from the output of the two types of crawlers, a pool of JAR files. The JARExtractor has taken all the 423 JAR files one by one, unpacked them, and extracted a few information via analyzing the ".txt" and ".MF" files of "META-INF" folder of those JAR files. The extracted information was the actual name of the JAR file, packages name and the implementation vendor details. The extracted information was a collection of words. Through this process, 2087 words were collected from those downloaded 423 JAR files. Those words were directly sent to the analyzing phase.

JAR files consist of binary files. The ASMExtractor was used to extract information (class names, class fields, method names, and method arguments) from binary (.class files) files. The module collected 9723 words. There were 2103 real meaningful words and 7620 connected words. The real meaningful words were directly sent to the analyzing process and the connected words were transferred to the ApacheLuceneSplitter, N-gramChunker, and WordIdentifier modules. Through those processes, 17227 words have been produced. All those words were used in the analyzer for ranking. Table III shows the details about those words' information.

The analyzer used WordNet to get synonym of all the words. In the ranking process, the words extracted from the given software architecture have been taken one by one and compared with the words extracted from the downloaded JAR files. If the words are matched with each other, marks will be assigned for that. If the words are not matched, the synonym of the words taken by using WordNet, and then do the same process.

The JAR files that achieved maximum marks have been selected through the processes mentioned above, which were 97 JAR files among 423 Downloaded JAR files. But the number of JAR files expected was 101, and the suggested number of JAR files was 97. The proposed framework failed to suggest 04 number of JAR files.

TABLE III. : DETAILS ABOUT DOWNLOADED JAR FILES

# of Downloaded JARs	# Words from META-INF folder	# of words from .class files	# of Real meaningful words	# of connected words, & Real words produced from them
423	2087	9723	2103	7620, 17227

V. DISCUSSION

Research implications. The findings of our research aim show that suggesting related JAR files when a software architecture enters into the development process is time-consuming and most difficult unless the developer is experienced and familiar with the context of the software project. The challenge mentioned above is very common for novice developers working in a complex and large software project. The proposed system is a very good solution for them, when using the proposed JarBot, the time required to find the necessary JAR files is reduced by automatically suggesting the necessary JAR files within a short period.

We hope our study will draw new directions for assessing the value of the search-based JAR file suggestions. In addition, the Apache Lucene and ASM , as used in JarBot, is more effective and efficient than using our earlier criteria (In our earlier work [5], our own developed camel case and explicit splitter were used to handle the connected words). A considerable performance increase is noticed in JarBot via using Apache Lucene and ASM in some modules.

Moreover, the JarBot helping the developer to complete the software project within the targeted time via automatically attach the necessary JAR files. And the other advantage is that software projects are being developed with quality assurance because JarBot will suggest the well completed and well-relevant JAR files from the selected websites. Therefore, JarBot can be highly recommended for novice developers who can work as trained and experienced software professionals, saving considerable time and money for the project which is large or small and complex or simple projects in which they work. The developers can benefit in another way using JarBot, i.e., by anticipating the required JAR files in advance (before the software architecture inserts into the JarBot) and comparing the incoming answer from the JarBot, and improving themselves.

Practical implications. In the software development process, JarBot can identify JAR files when giving a software architecture (class diagram) in XML format. Developers can make their programs more robust with this framework.

Developers can use this proposed framework to correctly suggest and handle the necessary JAR files within a short period than a developer takes by doing the same process manually. The evaluation phase of this study proved that the JarBot could suggest more than 95% of the necessary JAR files. Also, the testing phase of this study proved that all the modules of JarBot, such that XMLExtractor, crawlers, JARExtractor, ASMExtractor, ApacheLuceneSplitter, WordIinder, N-gramChunker, and Analyzer are working perfectly. Without the high accuracy of the modules mentioned above, it was impossible to suggest more than 95% of the expected JAR files by the JarBot.

The usage of the WordNet and the N-gram greatly reduces the chances of necessary JAR files going wrong and missing because the N-gram technique help to identify the real words from all the abbreviated identifiers. And the usage of WordNet provides a synset (a list of synonyms) for each word in the analyzing process. In this way, the probability of missing or going wrong of necessary JAR files was very less.

Limitation of this research. The main limitation of this work is that the proposed system is only suitable for Java programming language because it is fully based on JAR file (Java Achieve) suggestion. Another limitation of the JarBot is which starts the process with the software architecture, in this work, the class diagram is only used as software architecture, and other diagrams also can be used. But all the software architecture needs to be given in XML format. The next limitation of the work is that though there are several websites for JAR file download, the JarBot included only two websites, such as *java2s* and *jar-download.* The next limitation is what we say, even if some words have more than ten synonyms, by default, the WordNet will answer only ten synonyms. All the limitations mentioned above can be broken via future modification of the JarBot.

VI. CONCLUSION

Novice developers face difficulties when finding necessary JAR files for a Java software project when a software architecture enters into the development process. In this paper, we presented a framework, JarBot, to automatically suggest the necessary JAR files for given software architecture in XML format in the development process. The JarBot includes modular architecture with many components such as XMLExtractor, crawlers, JARExtractor, ASMSplitter, ApacheLuceneSplitter, WordIdentifier, N-gramChunker, and Analyzer. Each module mentioned above is interdependent, and the output of one module is input to another module. We validated the JarBot against 05 well-completed Java software projects, targeting used JAR files in those selected projects. The software architectures (class diagrams) for the selected 05 Java projects were designed and exported as XML files. And then, the software designs were given as input to the JarBot. The JarBot has suggested more than 95% of the expected JAR files. Our results show that JarBot efficiently suggests the necessary JAR files for software architecture in the development process.

In the future, we aim to extend JarBot along the following dimensions: (i) introduce support for all the diagrams as software architecture, (ii) to cover other all websites which are providing JAR files download, (iii) enable JarBot to use all the list of synonym of a particular word using in analyzing process.

REFERENCES

[1] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. Van Deursen, "Effective and efficient API misuse detection via exception propagation and search-based testing," ISSTA 2019 - Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal., pp. 192–203, 2019, doi: 10.1145/3293882.3330552.

[2] M. Lamothe and W. Shang, "Exploring the use of automated API migrating techniques in practice," Proc. 15th Int. Conf. Min. Softw. Repos. - MSR '18, pp. 503–514, 2018.

[3] N. Murakami and H. Masuhara, "Optimizing a search-based code recommendation system," 2012 3rd Int. Work. Recomm. Syst. Softw. Eng. RSSE 2012 - Proc., pp. 68–72, 2012, doi:

10.1109/RSSE.2012.6233414.

[4] M. Di Penta, D. M. German, and G. Antoniol, “Identifying licensing of jar archives using a code-search approach,” Proc. - Int. Conf. Softw. Eng., pp. 151–160, 2010, doi: 10.1109/MSR.2010.5463282.

[5] P. Pirapuraj and I. Perera, “Analyzing source code identifiers for code reuse using NLP techniques and WordNet,” 3rd Int. Moratuwa Eng. Res. Conf. MERCon 2017, no. May, pp. 105–110, 2017, doi: 10.1109/MERCon.2017.7980465.

[6] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, “Mining concepts from code with probabilistic topic models,” ASE’07 - 2007 ACM/IEEE Int. Conf. Autom. Softw. Eng., no. January, pp. 461–464, 2007, doi: 10.1145/1321631.1321709.

[7] A. T. Nguyen et al., “API code recommendation using statistical learning from fine-grained changes,” Proc. ACM SIGSOFT Symp. Found. Softw. Eng., vol. 13-18-Nove, pp. 511–522, 2016, doi: 10.1145/2950290.2950333.

[8] S. Vargas-Baldrich, M. Linares-Vásquez, and D. Poshyvanyk, “Automated tagging of software projects using bytecode and dependencies,” Proc. - 2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2015, pp. 289–294, 2016, doi: 10.1109/ASE.2015.38.

[9] A. T. Nguyen and T. N. Nguyen, “Graph-based statistical language model for code,” Proc. - Int. Conf. Softw. Eng., vol. 1, pp. 858–868, 2015, doi: 10.1109/ICSE.2015.336.

[10] X. Liu, L. G. Huang, and V. Ng, “Effective API recommendation without historical software repositories,” ASE 2018 - Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng., pp. 282–292, 2018, doi: 10.1145/3238147.3238216.

[11] H. Alrubaye, M. W. Mkaouer, and A. Ouni, “MigrationMiner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level,” Proc. - 2019 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2019, pp. 414–417, 2019, doi: 10.1109/ICSME.2019.00072.

[12] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, “NonDex: A tool for detecting and debugging wrong assumptions on Java api specifications,” Proc. ACM SIGSOFT Symp. Found. Softw. Eng., vol. 13-18-Nove, pp. 993–997, 2016, doi: 10.1145/2950290.2983932.

[13] “ASM.” [Online]. Available: https://asm.ow2.io/. [Accessed: 10-Feb-2021].

[14] “Apache Lucene - Welcome to Apache Lucene.” [Online]. Available: https://lucene.apache.org/. [Accessed: 10-Feb-2021].