UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Analyzing Outliers in the Maven Central Repository with Object Oriented Design Metrics

Permalink
https://escholarship.org/uc/item/93b235sg

Author
Kadge, Saumitra Prabhat

Publication Date
2020

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Analyzing Outliers in the Maven Central Repository with Object Oriented Design Metrics

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Saumitra Prabhat Kadge

Thesis Committee:
Professor Cristina Lopes, Chair
Assistant Professor Iftekhar Ahmed
Assistant Professor Joshua Garcia

2020

© 2020 Saumitra Prabhat Kadge

TABLE OF CONTENTS

LIST OF FIGURES .... **iv**

LIST OF TABLES .... **v**

ACKNOWLEDGEMENTS .... **vi**

ABSTRACT OF THE THESIS .... **vii**

Introduction **1**

1.1 Motivations 1

1.2 Object Oriented Design Metrics 1

1.2.1 Quality Model for Object Oriented Design (QMOOD) 1

1.2.2 Chidamber and Kemerer Object Oriented Metrics 4

1.3 Datasets of Design Metrics 7

1.3.1 Maven Dependency Dataset 7

1.3.2 Bug Catalog of the Maven Ecosystem 9

1.3.3 Metrics Dataset of the Java Ecosystem 10

Building the Dataset **11**

2.1 Building the Dataset 11

2.1.1 Maven Central Repository 11

2.1.2 Reading the Maven Central Index 12

2.1.3 Downloading project JAR files 13

2.2 Calculating the Design Metrics 16

2.2.1 CKJM-extended 17

2.2.2 JDepend 21

2.2.3 MariaDB database schema 23

Analysis of the Dataset **26**

3.1 Approach to analysis 26

3.2 Class level analysis 27

3.2.1 Size metrics (LOC, WMC, NPM, RFC and LCOM) 28

3.2.2 Coupling metrics (Ce, CBO, Ca and NOC) 31

3.2.3 Encapsulation and Cohesion metrics (DAM and LCOM3) 34

3.2.4 Inheritance metrics (DIT, MFA, IC and CBM) 36

3.2.5 Cohesion among methods of class (CAM) 39

3.2.6 Average method complexity (AMC) 40
3.2.7 Measure of aggregation (MOA) 40
3.3 Overall observations from class level analysis 41
3.4 Package level analysis 42
3.4.1 Size metrics (NC, NCC and NAC) 43
3.4.2 Efferent couplings (Ce) 45
3.4.3 Popularity metrics (Ca, $\boldsymbol{A}$, $\boldsymbol{I}$ and DMM) 46
3.5 Overall observations from package level analysis 49
Conclusion 50
4.1 Limitations 50
4.2 Future Work 51
4.3 Conclusion 51
Bibliography 52
Appendix 57

LIST OF FIGURES

Figure 2.1 – POM file template. 14
Figure 2.2 – Complete POM file for an artifact. 15
Figure 2.3 – CKJM-extended output. 20
Figure 2.4 – JDepend output 23
Figure 2.5 – Database schema. 24
Figure 3.1 – Correlation matrix for class level metrics. 28
Figure 3.2 – Lines of Code 30
Figure 3.3 – Weighted Methods per Class 30
Figure 3.4 – Number of Public Methods 30
Figure 3.5 – Response for a Class 31
Figure 3.6 – Lack of Cohesion in Methods 31
Figure 3.7 – Efferent Couplings 33
Figure 3.8 – Coupling Between Object Classes 33
Figure 3.9 – Afferent Couplings 33
Figure 3.10 – Number of Children 34
Figure 3.11 – Data Access Metric 35
Figure 3.12 – Lack of Cohesion in Methods 3 35
Figure 3.13 – Depth of Inheritance Tree 37
Figure 3.14 – Measure of Functional Abstraction 38
Figure 3.15 – Inheritance Coupling 38
Figure 3.16 – Coupling Between Methods 38
Figure 3.17 – Cohesion Among Methods of Class 39
Figure 3.18 – Average Method Complexity 40
Figure 3.19 – Measure of Aggregation 41
Figure 3.20 – Correlation matrix for package level metrics. 43
Figure 3.21 – Number of Classes 44
Figure 3.22 – Number of Concrete Classes 45
Figure 3.23 – Number of Abstract Classes 45
Figure 3.24 – Efferent Couplings 46
Figure 3.25 – Afferent Couplings 47
Figure 3.26 – Abstractness 48
Figure 3.27 – Instability 48
Figure 3.28 – Distance Main Measure 48

LIST OF TABLES

Table 1.2.1 – Design quality attributes 2
Table 1.2.2 – Design properties 3
Table 1.2.3 – Design metrics 4
Table 1.2.4 – Design quality attributes derived from design properties 4
Table 2.1 – Maven Central Index fields. 13
Table 2.2 – Statistics of downloaded JAR files. 16
Table 2.3 – Statistics of JARs with and without classes. 16
Table 2.4 – CKJM-extended field descriptions. 19
Table 2.5 – CKJM analysis results. 21
Table 2.6 – JDepend field descriptions. 22
Table 2.7 – JDepend analysis statistics. 22
Table 2.8 – Database table significance. 25

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Cristina Lopes for her guidance and support at every stage of this project. I would also like to thank my parents for their encouragement.

ABSTRACT OF THE THESIS

Analyzing Outliers in the Maven Central Repository with Object Oriented Design Metrics

by

Saumitra Prabhat Kadge

Master of Science in Software Engineering

University of California, Irvine, 2020

Professor Cristina Lopes, Chair

Software quality is increasingly becoming a differentiator between software products. This resulted in the development of new and improved approaches to software development like object-orientation and the development of software metrics to better manage the process of software development. Analyzing a large collection of software projects that use object-oriented programming in terms of object-oriented design quality metrics that measure the size, complexity, performance and quality of software could give us a good idea of how object-oriented programming is used in practice. Analyzing software projects on the extreme ends of the distributions of these metrics could give us specific examples of coding practices which influence design qualities. This analysis can be used to inform machine learning models that estimate code quality based on design metrics.

In this thesis, I generated a repository containing the latest version of 226,793 software projects taken from the Maven Central Repository. I statically analyzed each of these projects and measured 18 class-level design quality metrics for 10,608,920 classes and 8 package-level design quality metrics for 2,107,577 packages. I analyze the outlier projects in terms of object-oriented design quality metrics and evaluate their suitability for inclusion in training sets for machine learning models.

Chapter 1

Introduction

1.1 Motivations

Software quality is increasingly becoming a differentiator between software products. This resulted in the development of new and improved approaches to software development like object-orientation and the development of software metrics to better manage the process of software development. Analyzing a large collection of software projects that use object-oriented programming in terms of object-oriented design quality metrics that measure the size, complexity, performance and quality of software could give us a good idea of how object-oriented programming is used in practice. Analyzing software projects on the extreme ends of the distributions of these metrics could give us specific examples of coding practices which influence design qualities. This analysis can be used to inform machine learning models that estimate code quality based on design metrics.

1.2 Object Oriented Design Metrics

1.2.1 Quality Model for Object Oriented Design (QMOOD)

Bansiya and Davis [1] developed Quality Model for Object Oriented Design (QMOOD) as a hierarchical model to measure the design quality attributes of object-oriented designs. The development of their hierarchical model was driven by three needs. First was the need for a way to measure software quality of object-oriented designs in the early stages of the software development cycle. Second was the need for a way to relate measurable object-oriented design properties like coupling, cohesion, encapsulation, inheritance, polymorphism etc. with desirable software quality attributes like reusability, effectiveness etc. Third was the need to have a process of measurement that could be automated and was non-intrusive.

Previously developed software quality models [2] assumed that internal product characteristics influence external product attributes. Thus, by evaluating the internal characteristics, conclusions can be drawn about the external attributes. McCall [3] had developed a model that defined software qualities as a hierarchy of factors, criteria and metrics and was one of the first hierarchical models. International efforts also resulted in the ISO 9126 standard for software product quality measurement. Bansiya and Davis identified two problems with these early models. First was that these models were vague about the details and metrics needed to assess product quality. Second problem was that these models could not account for the conflicting ways in which quality attributes could influence overall software quality.

Dormey [4] proposed a framework to develop software quality models which involves breaking down design quality attributes into measurable design properties. Bansiya and Davis used this methodology in the development of Quality Model for Object-Oriented Design (QMOOD). They start by defining an initial set of design quality attributes which are abstract concepts. Table 1.2.1 shows these design quality attributes.

Quality attribute	Meaning
Reusability	Characteristics that allow a design to be applied to a new problem.
Flexibility	Characteristics that allow changes to be made in a design.
Understandability	Characteristics that allow a design to be comprehended.
Functionality	Responsibilities assigned to the classes of a design that are available through public interfaces.
Extendibility	Characteristics that allow the addition of new requirements in a design.
Effectiveness	Ability of a design to achieve desired functionality and behavior.

Table 1.2.1 – Design quality attributes

They then define a set of measurable design properties which can be calculated by looking at the structure of classes, attributes and methods. Table 1.2.2 shows these design properties.

Design property	Meaning
Design Size	Measure of the number of classes used in a design.
Hierarchies	Measure of the number of non-inherited classes that have children in a design.
Abstraction	Measure of the generalization-specialization aspect of a design.

Encapsulation	Protecting the internal representation of objects by defining attribute declarations as private in a design.
Coupling	Number of other objects that would have to be accessed by an object for it to function correctly in a design.
Cohesion	Strong overlap in method parameters and attribute types indicates strong cohesion.
Composition	Measure of 'part-of', 'has', 'consists-of' or 'part-whole' relationships in a design.
Inheritance	Measure of 'is-a' relationship between classes.
Polymorphism	Measure of the services in an object that are dynamically determined at runtime.
Messaging	Measure of the services that a class provides to other classes.
Complexity	Degree of difficulty in comprehending the internal and external structure of classes and their relationships.

Table 1.2.2 – Design properties

Each of the design properties defined previously can be evaluated objectively using one of the design metrics. Table 1.2.3 shows all the design metrics and the corresponding design property they measure.

Metric	Description	Design property
Design Size in Classes (DSC)	Number of classes in a design.	Design Size
Number of Hierarchies (NOH)	Number of class hierarchies in a design.	Hierarchies
Average Number of Ancestors (ANA)	Average number of classes from which a class inherits information.	Abstraction
Data Access Metric (DAM)	Ratio of the number of private attributes to the total number of attributes declared in a class.	Encapsulation
Direct Class Coupling (DCC)	Number of classes that a class is directly related to by attribute declaration and method parameters.	Coupling
Cohesion Among Methods of Class (CAM)	Summation of the intersection of parameters of a method with the maximum independent set of all parameter types in a class.	Cohesion
Measure of Aggregation (MOA)	Number of data declarations whose types are user defined classes.	Composition
Measure of Functional Abstraction (MFA)	Ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class.	Inheritance
Number of Polymorphic Methods (NOP)	Number of methods that can exhibit polymorphic behavior.	Polymorphism

Class Interface Size (CIS)	Number of public methods in a class.	Messaging
Number of Methods (NOM)	Number of all methods defined in a class.	Complexity

Table 1.2.3 – Design metrics

Lastly, they weighted the contribution of individual design properties in influencing design quality attributes. Table 1.2.4 shows the contributions of design properties to calculate design quality attributes.

Quality attribute	Equation in terms of design properties
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation – 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation – 0.33 * Coupling + 0.33 * Cohesion – 0.33 * Polymorphism – 0.33 * Complexity – 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Table 1.2.4 – Design quality attributes derived from design properties

Bansiya and Davis also evaluated their metrics suite on different versions of Microsoft Foundation Classes (MFC) and Borland Windows Object Library (OWL) to study the variation in design quality attributes with versions. They found a decrease in understandability in the initial versions of these frameworks as more functionality was added with every version. As the frameworks matured, understandability remained stable. The scores for reusability, flexibility, functionality, extendibility and effectiveness went on increasing with every version. The QMOOD metrics suite was also validated on a set of 14 medium sized C++ projects. The rankings given by 13 human evaluators and those given by QMOOD were found to correlate significantly. I used the CKJM-extended [5] tool to calculate the DAM, MOA, MFA, CAM and CIS metrics.

1.2.2 Chidamber and Kemerer Object Oriented Metrics

Chidamber and Kemerer [6] developed a suite of six metrics for object-oriented design. Their investigation of previous research in software metrics showed that it either lacked a theoretical basis, or lacked useful measurement properties, or was not adequately generalized, or depended heavily on the implementation, or the metrics developed were difficult to collect. They based their metrics on the ontology of Bunge [7] [8]. Per this ontology, the world is made up of 'things' called as 'substantial individuals' which possess properties. While 'properties' are inherent to 'substantial individuals', an observer can assign features to an individual called as attributes. A 'substantial individual' and its 'properties' collectively form an 'object'. The ontology also gives the general principles of coupling and cohesion. Good software design should minimize coupling i.e. the interdependence between parts of a design; and maximize cohesion i.e. the consistency within parts of a design.

A need was felt to develop software metrics with a greater degree of theoretical and mathematical rigor. While traditional approaches to software development take a function-oriented view which separates data from the procedures that operate on it, the object-oriented approach involves modeling the real world in terms of objects. It was found that software metrics developed for the traditional approach could not be readily used for the object-oriented approach [9]. Several object-oriented design methodologies including the one proposed by Booch [10] consider class design as the most important in object-oriented design. The Chidamber and Kemerer metrics were thus developed to measure the design complexity of classes in an implementation independent manner.

To evaluate their metrics, they used six of the nine properties in a formal list of criteria developed by Weyuker [11]. The properties that were used for analysis are listed below.

1. **Non-coarseness**: Not every class should have the same value for a metric.
2. **Non-uniqueness**: It is possible for two classes to have the same value for a metric.
3. **Design details are important**: Even if two classes perform the same function, their metric values should be determined by the details of their design.
4. **Monotonicity**: The metric value for a combination of two classes can never be less than the metric values of either of the component classes.

5. **Non-equivalence of interaction**: For classes P, Q and R where classes P and Q have the same metric value, it is possible that the metric value of the combination of classes P and R is different from the metric value of the combination of classes Q and R.
6. **Interaction increases complexity**: For classes P and Q the sum of the metric values of P and Q should be less than the metric value of their combination.

The metrics developed by Chidamber and Kemerer and their significance are described in the sections below.

1. **Weighted Methods per Class (WMC)**: Sum of the complexities of all methods of a class. If complexity is considered as unity for every method in a class, then this metric equals the total number of methods in that class.
   Significance: The number of methods in a class and their complexities can be used as an indicator of the time required to develop and maintain it. A class with many methods could have a significant impact on child-classes which would inherit its methods. A class with many methods could also mean that it is specialized thus limiting its reuse.
2. **Depth of Inheritance Tree (DIT)**: Maximum length from a class node to the root of the class hierarchy tree.
   Significance: A class that is deep in the class hierarchy tree is likely to inherit many methods. This could result in a greater reuse of the inherited methods. It could also increase the difficulty of predicting its behavior in response to messages. Deeper class inheritance trees result in greater design complexity. Looking at this metric it is possible to determine if a given design has too many classes close to/farther from the root of the hierarchy tree.
3. **Number of Children (NOC)**: Number of immediate subclasses of a class.
   Significance: More the number of immediate subclasses of a class, greater is the extent of reuse of its methods. This could result in a greater influence of that class on the overall design and thus increase the importance of testing the methods of that class. A large value for this metric may also indicate a problem with subclassing of a class due to improper abstraction.
4. **Coupling Between Object classes (CBO)**: Number of other classes to which a class is coupled.

Significance: To promote modularity and reuse of code it is desirable to have minimum coupling between classes. Greater the coupling, more is the sensitivity to changes in design. A higher value for this metric could indicate the need for more complex testing. This metric can also be tracked by designers to check if the system is developing any unnecessary interconnections.

5. **Response for a Class (RFC)**: Union of the set of methods in a class with sets of methods called by each of them.

   Significance: A higher value for this metric for a class means that a large number of methods could be invoked in response to a message received by an object of that class. This means the class has more complexity and the testing and debugging would require more effort.

6. **Lack of Cohesion in Methods (LCOM)**: Consider all pairs of methods of a class. Lack of cohesion in methods is the difference between the size of set formed by all method pairs which do not share any instance variable and the size of set formed by all method pairs which share at least one instance variable.

   Significance: This metric can be used to identify classes with disparate methods that try to do different things and split them into subclasses. Lower value of cohesion increases complexity.

All the six metrics defined above were found to satisfy a majority of Weyuker's properties except the sixth which was not satisfied by any metric. To calculate the Chidamber and Kemerer metrics for my purposes I used the CKJM-extended [5] tool.

1.3 Datasets of Design Metrics

In the following sections I look at various attempts to create datasets of metrics from the Maven Central Repository.

1.3.1 Maven Dependency Dataset

Raemaekers et al [12] built the Maven Dependency Dataset (MDD) with the goal of facilitating large scale research on software releases, versions and evolving dependencies at the level of packages, classes and methods. They took a snapshot of the Maven Central Repository on July 30, 2011 consisting of 148,253 source and binary JAR files. The snapshot was found to contain 22,111 projects with an average of 6.7 versions per project. They collected a large amount of data from these JAR files which they split and stored into a MySQL database, a Berkeley database and a Neo4j graph database.

The MySQL database contained four major tables along with a few other supporting tables. The 'files' table had metrics like number of methods, number of methods compared to the next version etc. The 'stats' table had metrics like lines of code, McCabe's cyclomatic complexity, number of methods, number of classes etc. The 'changes' table stored both breaking and non-breaking changes between library versions along with the names of affected packages, classes, methods and attributes for every change. The 'deps' table stored all library dependencies present in the build configuration files of projects along with an 'isolation rating' for every dependency which gave the percentage of files which did not import it.

The Berkeley database being an on-disk key-value store, enabled the fast lookup of information about individual methods, classes and packages. Information about 36,695,764 unique methods, classes and packages was stored in this database. This database can be used to obtain a list of all methods in a specific version of a library.

The Neo4j database stored call graph information and could be queried with the Cypher query language. The call graph comprised of methods, classes, packages and JAR files as 'Units' where 'Units' are connected by either method call, inheritance, historical or containment relationships. The graph was stored as a collection of tuples where every tuple comprised of two 'Unit Identifiers' connected with any one of the four relationship types mentioned previously. This database can be used to track changes in a specific library through time.

To process this large number of JAR files, the DAS-3 supercomputer [13] was used with the central node holding the databases and compute nodes doing the computation. One of the

limitations of this research is that any updates to the Maven Central Repository after the snapshot was taken were not considered. My research differs from this in that not only do I consider size related metrics, but I also compute and store object-oriented design quality metrics such as those from the Chidamber-Kemerer and QMOOD metric suites.

1.3.2 Bug Catalog of the Maven Ecosystem

Mitropoulos et al [14] built a dataset by statically analyzing the Maven Central Repository using FindBugs and storing the resulting FindBugs metrics along with size and dependency metadata for every JAR file in the repository. Their snapshot of the Maven Central Repository taken in January 2012 was approximately 256GB in size, consisted of 115,214 JAR files that made up 17,505 projects with a mean of 6.58 versions for every project.

From the snapshot of the Maven Central Repository, they were able to obtain a list of every project version in the repository. They then filtered out projects written in languages other than Java and the resulting list was used to create a series of processing tasks with one task corresponding to every project version. These tasks were added to a RabbitMQ queue mechanism and 25 Python threads were started which would check out processing tasks form the queue, process them and store the results in a MongoDB database. Every processing task involved calculating and storing JAR file metadata of the corresponding project version, checking for the presence of .class files in the JAR file, executing FindBugs and storing the results in the database.

For every JAR, FindBugs reports all the bugs present in that JAR along with the category (nine categories), class, method, and line for every bug found. FindBugs also reports additional metadata like the number of classes etc. for every JAR. In addition to these metrics reported by FindBugs, the Python threads also calculated other metadata for every JAR file such as size, dependencies (taken from pom.xml) and version number (taken from maven-metadata.xml).

The dataset was used to calculate the correlation between defect count and project version number, the correlation between defect count and project version size and the pairwise

correlations between defect categories. This dataset also has the issue of getting older as newer project versions are released.

1.3.3 Metrics Dataset of the Java Ecosystem

Karakoidas et al [15] built a dataset by statically analyzing the Maven Central Repository using CKJM [5], JDepend [16] and CLMT [17] tools. Their snapshot of the Maven Central Repository consisted of 22,730 JAR files that made up 11,365 projects.

From their initial snapshot of the Maven Central Repository, they considered only Java projects with the latest project version. In addition to this, they also filtered out those projects which did not have both binary and source JARs. The rest of the process to build the dataset was like that used by Mitropoulos et al [14] with the only difference being the tools used to calculate metrics. The results were stored in a MySQL database and were used in experiments to determine the Domain-Specific Language (DSL) usage in Java projects.

I take a similar approach to that taken by Karakoidas et al. However, instead of filtering out projects lacking both binary and source JAR files, I consider all projects with a binary JAR file and calculate object-oriented design quality metrics for all of them. I also consider JAR files with code written in languages that compile to Java bytecode like Scala, Groovy, Kotlin, Clojure etc. I also work with a much larger set of JAR files taken from the Maven Central Repository.

Chapter 2

Building the Dataset

2.1 Building the Dataset

2.1.1 Maven Central Repository

Apache Maven is a build automation tool used to manage the process of building software projects and managing their dependencies [18]. Although primarily used for Java projects, it can also be used for Scala, Ruby, C# and other languages. Maven uses the Project Object Model to provide general project configuration such as project name, owner, dependencies etc. It can also include configurations for plugins. Most Maven functionality is provided by plugins which provide a set of goals. It is possible to do things like compile Java projects using the 'compile' goal of the 'compiler' plugin, test source code using the 'test' goal of the 'surefire' plugin, or package sources and resources into a JAR file using the 'jar' goal of the 'jar' plugin. To avoid running each of these goals manually, Maven introduces build lifecycles which are lists of named phases. Every phase can be associated with a goal provided by a plugin. For example, the 'compile' goal of the 'compiler' plugin can be associated with the 'compile' phase of the 'default lifecycle'. When a command like 'mvn test' is executed, all goals associated with all phases before and including the 'test' phase are executed.

Dependency management is an important feature provided by Maven. The Project Object Model of a project can be used to specify its dependencies. A coordinate system is used to identify these dependencies which can be artifacts like software libraries or modules. Maven automatically downloads all dependencies and their transitive dependencies and stores them in a local repository. By default, the Maven 2 Central Repository is used to search for dependencies. As of this writing, the Maven Central Repository holds 5,745,274

JAR files [19]. Thus, the JAR files from the Maven Central Repository can be a good source to obtain object-oriented design metrics for the Java ecosystem.

2.1.2 Reading the Maven Central Index

The Maven Central Repository also provides the Maven Central Index [20] in gzip file format. This index is updated weekly and is available for download as a whole or as weekly increments. The index is built using Maven Indexer and offers various fields containing information about all artifacts in the Maven Central Repository. Table 2.1 shows all the fields present in the Maven Central Index [21].

Indexer type	Index field	Significance
-	u:	Artifact groupId|artifactId|version|classifier|extension/packaging (keyword, stored)
min	i:	Artifact packaging|lastModified|size|sourcesExists|javadocExists|signatureExists (not indexed, stored)
	g:	Artifact GroupID (keyword)
	groupId:	Artifact GroupID (tokenized)
	a:	Artifact ArtifactID (keyword)
	artifactId:	Artifact ArtifactID (tokenized)
	v:	Artifact Version (keyword)
	version:	Artifact Version (tokenized)
	p:	Artifact packaging/extension (keyword)
	l:	Artifact classifier (keyword)
	n:	Artifact name (tokenized, stored)
	d:	Artifact description (tokenized, stored)
	m:	Artifact last modified (not indexed, stored)
	1:	Artifact SHA1 checksum (keyword, stored)
jarContent	classnames:	Artifact Classes (tokenized)
	c:	Artifact Classes (tokenized on newlines)
maven-plugin	px:	MavenPlugin prefix (keyword, stored)
	gx:	MavenPlugin goals (keyword, stored)
maven-archetype	No field	
osgi-metadatas	Bundle-SymbolicName:	Bundle-SymbolicName (indexed, stored)
	Bundle-Version:	Bundle-Version (indexed, stored)
	Export-Package:	Export-Package (indexed, stored)

	Export-Service:	Export-Service (indexed, stored)
	Bundle-Description:	Bundle-Description (indexed, stored)
	Bundle-Name:	Bundle-Name (indexed, stored)
	Bundle-License:	Bundle-License (indexed, stored)
	Bundle-DocURL:	Bundle-DocURL (indexed, stored)
	Import-Package:	Import-Package (indexed, stored)
	Require-Bundle:	Require-Bundle (indexed, stored)

Table 2.1 – Maven Central Index fields.

I started by downloading the gzip file of the latest complete Maven Central Index and used the Maven Indexer CLI [22] to unpack it to a Lucene index in a directory. I then used Luke [23] which is a Lucene Index browser to examine the Maven Central Index. From Table 2.1 the 'u' field gives a concatenated string containing the Group ID, Artifact ID and version of every artifact in the Maven Central Repository. I examined this field using Luke and found values like 'abbot|abbot|0.12.3|NA', 'abbot|abbot|0.13.0|NA', 'abbot|abbot|1.4.0|NA', 'abbot|abbot|1.4.0|javadoc|jar', 'abbot|abbot|1.4.0|sources|jar' etc. I then wrote a Java program that would sort the entries in this field and write them to text files in chunks of 10,000 entries per text file. While writing these entries to the text files, the program would skip entries ending with 'javadoc|jar' and 'sources|jar' as these would correspond to JAR files containing documentation or source code which I did not need for my analysis. The program also used the 'ComparableVersion' class provided by 'Maven Artifact v3.0.3' [24] to compare different artifact versions with the same Group ID and Artifact ID and eliminated all versions other than the latest version. As a result, I was left with text files containing a 'GroupID|ArtifactID|Version' entry for every unique artifact in the Maven Central Repository. Every text file had up to 10,000 entries, but the average number of entries per text file was much less because of the eliminations mentioned previously. The chunking was done so that I could run multiple threads and download multiple artifacts at the same time without having multiple threads read from the same file.

2.1.3 Downloading project JAR files

From the previous step I had the Group ID, Artifact ID and Version coordinates for every unique artifact in the Maven Central Repository written to text files. I then used Maven Dependency Plugin [25] to download all the JAR files from the Maven Central Repository to

a local cache directory and then copy them to my repository with a directory structure that matched the GroupID, ArtifactID and Version of the downloaded JARs.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <project xmlns="http://maven.apache.org/POM/4.0.0"
 3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 5     <modelVersion>4.0.0</modelVersion>
 6 
 7     <groupId>jarDownloader</groupId>
 8     <artifactId>jarDownloader</artifactId>
 9     <version>1.0-SNAPSHOT</version>
10 
11     <build>
12         <plugins>
13             <plugin>
14                 <groupId>org.apache.maven.plugins</groupId>
15                 <artifactId>maven-dependency-plugin</artifactId>
16                 <version>3.1.1</version>
17                 <executions>
18                     <execution>
19                         <id>copy</id>
20                         <phase>package</phase>
21                         <goals>
22                             <goal>copy</goal>
23                         </goals>
24                     </execution>
25                 </executions>
26                 <configuration>
27                     <artifactItems>
28                     </artifactItems>
29                     <outputDirectory>./DownloadedJARsLatestVersion</outputDirectory>
30                     <overWriteReleases>false</overWriteReleases>
31                     <overWriteSnapshots>true</overWriteSnapshots>
32                 </configuration>
33             </plugin>
34         </plugins>
35     </build>
36 
37 </project>
```

Figure 2.1 – POM file template.

To use the Maven Dependency Plugin, I started by creating a template POM file with the required configuration. Figure 2.1 shows the template POM file. Note that the Maven Dependency Plugin has been configured with 'copy' as the goal and it has been associated with the 'package' phase of the build lifecycle. The 'copy' goal is used to resolve the artifacts configured inside the 'artifactItems' element of the POM file and place them in a configurable directory [26] [27]. For every entry of GroupID, ArtifactID and Version in the text files generated previously, I added an 'artifactItem' element to the template POM file to

get a complete POM file that could be used by Maven. Figure 2.2 shows one such complete POM file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>jarDownloader</groupId>
    <artifactId>jarDownloader</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>3.1.1</version>
                <executions>
                    <execution>
                        <id>copy</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <artifactItems>
                      <artifactItem>
                        <groupId>software.amazon.awssdk</groupId>
                        <artifactId>alexaforbusiness</artifactId>
                        <version>2.9.9</version>
                        <type>jar</type>
                        <outputDirectory>./DownloadedJARsLatestVersion/software/amazon/awssdk/alexaforbusiness/2.9.9</outputDirectory>
                      </artifactItem>
                    </artifactItems>
                    <outputDirectory>./DownloadedJARsLatestVersion</outputDirectory>
                    <overWriteReleases>false</overWriteReleases>
                    <overWriteSnapshots>true</overWriteSnapshots>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Figure 2.2 – Complete POM file for an artifact.

Once a complete POM file was generated, I executed the ‘copy’ goal and this would fetch the configured artifact from the Maven Central Repository and store it in a directory with a path that matched the GroupID, ArtifactID and Version of the retrieved artifact. I used Maven Invoker [28] to programmatically execute the ‘package’ phase in order to execute the ‘copy’ goal.

All the steps in this section were executed by a single Java program running on a machine with an Intel Xeon X5675 CPU with 24 cores and 125GiB memory. Initially I tried executing all steps in a single-threaded program, but it was taking very long to download all JARs and the CPU was largely unutilized. I then reimplemented the program to make it multi-

threaded. After decoding the Maven Central Index, it was split into several text files each holding up to 10,000 artifact coordinates. I then started 120 threads with each thread reading a range of text files and downloading the included artifacts. Every thread was assigned a number and the range of text files that it would read was based on this number. This prevented multiple threads from reading the same text file and allowed multiple threads to download artifacts at the same time. Also, every thread generated its own copy of the POM file template, and for every artifact coordinate that it would read from the text files, it would modify its own copy of the template. After implementing multi-threading, I saw significant improvements in the artifact retrieval time, and I was able to complete downloading all artifacts in 6 days. Table 2.2 shows some statistics related to the download.

Number of JAR files downloaded	269,211
Total size of JAR files downloaded	219.911 GB

Table 2.2 – Statistics of downloaded JAR files.

After downloading all JAR files, I scanned them with a Python script that examined every JAR file for the presence of class files. JAR files that did not contain any class files were moved to a different directory and were not analyzed further. Table 2.3 shows the details of the remaining JAR files.

JARs with class files	226,793
JARs without class files	38,568
Total size of JARs with class files	191.087 GB

Table 2.3 – Statistics of JARs with and without classes.

I ran two static analysis tools on the remaining JAR files to gather object-oriented design metrics. Following sections give more details of the analysis.

2.2 Calculating the Design Metrics

To calculate the object-oriented design metrics for all the downloaded JARs, I used CKJM-extended and JDepend. To execute each of these tools and save their results, I ran two Python scripts: an 'executor' script and a 'database' script. The 'executor' script would scan through the downloaded JAR repository and for every JAR file in the repository, it would

add the path of that JAR file to a queue. The script would then start 24 threads running in parallel and reading JAR file paths from the queue. Each of these threads analyze one JAR file at a time and store the result of analysis in either a text file (in case of CKJM-extended) or in an XML file (in case of JDepend). Thus, for every JAR file, one text file and one XML file were generated and stored in the CKJM and JDepend result directories. Once the 'executor' script finished execution, I ran the 'database' script. The 'database' script would scan through the CKJM/JDepend result directory and for every text/XML file in the result directory, it would add the path of that text/XML file to a queue. In a manner like the 'executor' script, the 'database' script would also start 24 threads running in parallel and reading text/XML file paths from the queue. Each of these threads parse one text/XML file at a time and store the extracted results in a MariaDB database. More details about each tool and the schema used for the MariaDB database are presented in the following sections.

2.2.1 CKJM-extended

CKJM-extended is a tool that can calculate size and object-oriented design metrics by processing bytecode of compiled Java files [5] [29] [30]. The metrics include all six metrics from the Chidamber and Kemerer metrics suite, six metrics from the QMOOD metrics suite and a few software quality-oriented extensions to the Chidamber and Kemerer metrics suite. Table 2.4 shows all the metrics calculated for every class.

Name of the metric	Metric description
Weighted methods per class (WMC)	Sum of the complexities of the methods of a class. Complexity is assigned as 1 to every method in a class. As a result, WMC of a class equals the number of methods in that class. (C&K metric) (Number of methods in QMOOD)
Depth of inheritance tree (DIT)	Number of inheritance levels from the top of the object hierarchy. (C&K metric)
Number of children (NOC)	Number of immediate descendants of a class. (C&K metric)
Coupling between object classes (CBO)	Number of classes coupled to a given class through inheritance, field access, method calls, arguments, return types and exceptions.

	(C&K metric)
Response for a class (RFC)	Sum of the number of methods called in the method bodies of a class and the number of methods in that class. (C&K metric)
Lack of cohesion in methods (LCOM)	Difference between the number of method pairs which do not share any field and the number of method pairs which share at least one field. (C&K metric)
Afferent couplings (Ca)	Number of other classes which use a given class.
Efferent couplings (Ce)	Number of other classes used by a given class.
Number of public methods (NPM)	Number of public methods of a class. (Class interface size in QMOOD)
Lack of cohesion in methods (LCOM3)	Lack of cohesion in methods is given by, $$LCOM3 = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$ Where m = number of methods a = number of attributes $\mu(A)$ = number of methods that access attribute A Constructors and static initializations are considered as separate methods. (Henderson-Sellers version)
Lines of code (LOC)	Number of lines of code in a class.
Data access metric (DAM)	Ratio of the number of private attributes in a class to the total number of attributes in that class. (QMOOD metric)
Measure of aggregation (MOA)	Number of attributes of a class whose types are user defined classes. (QMOOD metric)
Measure of functional abstraction (MFA)	Ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of that class. (QMOOD metric)
Cohesion among methods of a class (CAM)	Cohesion among methods of a class is given by, $$CAM = \frac{\sum_{j=1}^{m} P_j}{m * p}$$ Where m = number of methods p = number of unique method parameters in whole class P_j = number of unique method parameters in method j Preferred value is close to 1.0. (QMOOD metric)
Inheritance coupling (IC)	Number of parent classes to which a given class is coupled where coupling is defined by the conditions below.

	• An inherited method calls a redefined method. • An inherited method is called by a redefined method and uses a parameter defined in the redefined method. • An inherited method uses a variable defined in a new/redefined method.
Coupling between methods (CBM)	Number of new/redefined methods to which all the inherited methods are coupled.
Average method complexity (AMC)	Average number of Java binary codes in all methods of a class.

Table 2.4 – CKJM-extended field descriptions.

I ran CKJM-extended for every JAR file in the downloaded JAR repository using the 'executor' Python script mentioned previously. The tool can analyze multiple JAR files at the same time and return a single output for all of them. However, I ran the tool one JAR file at a time to separate the analysis results of every JAR file. The tool returns a textual output as shown in figure 2.3. I saved this textual output in text files separately for every JAR file. The output consists of a new line for every class and method in a package. For every class in the package, its fully qualified name is printed on a new line followed by the values of all the metrics separated by spaces. For every class, the tool also prints out the method signatures for all methods in that class. Every method signature is preceded by a '~' symbol and is followed by the value of the McCabe's cyclomatic complexity for that method. Keeping the analysis results separate for each JAR allowed me to associate the class names with their container JAR.

```
com.amazonaws.services.lambda.runtime.RequestStreamHandler 1 1 0 1 1 0 0 1 1 2.0000 1 0.0000 0 0.0000 1.0000 0 0 0.0000
 ~ public abstract void handleRequest(java.io.InputStream, java.io.OutputStream, com.amazonaws.services.lambda.runtime.Context): 1

com.amazonaws.services.lambda.runtime.LambdaRuntime 3 1 0 2 5 1 0 2 1 0.5000 14 1.0000 1 0.0000 0.5000 0 0 3.3333
 ~ private void <init>(): 1
 ~ static void <clinit>(): 1
 ~ public static com.amazonaws.services.lambda.runtime.LambdaLogger getLogger(): 1

com.amazonaws.services.lambda.runtime.LambdaLogger 2 1 0 3 2 1 3 0 2 2.0000 2 0.0000 0 0.0000 0.6667 0 0 0.0000
 ~ public abstract void log(byte[]): 1
 ~ public abstract void log(String): 1

com.amazonaws.services.lambda.runtime.LambdaRuntime$1 3 1 0 2 7 3 1 1 2 2.0000 18 0.0000 0 0.0000 0.5556 0 0 5.0000
 ~ void <init>(): 1
 ~ public void log(String message): 1
 ~ public void log(byte[] message): 1

com.amazonaws.services.lambda.runtime.Context 11 1 0 5 11 55 2 3 11 2.0000 11 0.0000 0 0.0000 1.0000 0 0 0.0000
 ~ public abstract String getLogStreamName(): 1
 ~ public abstract com.amazonaws.services.lambda.runtime.ClientContext getClientContext(): 1
 ~ public abstract String getAwsRequestId(): 1
 ~ public abstract String getFunctionVersion(): 1
 ~ public abstract com.amazonaws.services.lambda.runtime.LambdaLogger getLogger(): 1
 ~ public abstract String getFunctionName(): 1
 ~ public abstract int getRemainingTimeInMillis(): 1
 ~ public abstract int getMemoryLimitInMB(): 1
 ~ public abstract String getLogGroupName(): 1
 ~ public abstract com.amazonaws.services.lambda.runtime.CognitoIdentity getIdentity(): 1
 ~ public abstract String getInvokedFunctionArn(): 1

com.amazonaws.services.lambda.runtime.Client 5 1 0 1 5 10 1 0 5 2.0000 5 0.0000 0 0.0000 1.0000 0 0 0.0000
 ~ public abstract String getAppVersionCode(): 1
 ~ public abstract String getInstallationId(): 1
 ~ public abstract String getAppVersionName(): 1
 ~ public abstract String getAppPackageName(): 1
 ~ public abstract String getAppTitle(): 1

com.amazonaws.services.lambda.runtime.LambdaRuntimeInternal 3 1 0 0 4 1 0 0 2 0.5000 12 1.0000 0 0.0000 0.3333 0 0 2.6667
 ~ private void <init>(): 1
 ~ public static boolean getUseLog4jAppender(): 1
 ~ public static void setUseLog4jAppender(boolean useLog4j): 1

com.amazonaws.services.lambda.runtime.ClientContext 3 1 0 2 3 3 1 1 3 2.0000 3 0.0000 0 0.0000 1.0000 0 0 0.0000
 ~ public abstract java.util.Map getCustom(): 1
 ~ public abstract com.amazonaws.services.lambda.runtime.Client getClient(): 1
 ~ public abstract java.util.Map getEnvironment(): 1

com.amazonaws.services.lambda.runtime.RequestHandler 1 1 0 1 1 0 0 1 1 2.0000 1 0.0000 0 0.0000 1.0000 0 0 0.0000
 ~ public abstract Object handleRequest(Object, com.amazonaws.services.lambda.runtime.Context): 1
```

Figure 2.3 – CKJM-extended output.

After this I used the ‘database’ Python script to read every output text file and store the JAR file names, their constituent class names and the metric values for every class in a MariaDB database. I did not save the method signatures or the McCabe’s cyclomatic complexity values, although the schema of the database does allow doing so.

I used CKJM-extended release 2.3 and Java 15 (OpenJDK 15) in my analysis. When CKJM analyzes a JAR file, it requires all the dependencies of that JAR file to be present in the Java classpath. To analyze as many JAR files as possible I added the top 100 popular JAR files from the Maven Central Repository to the Java classpath along with the JAR file of CKJM-extended. For every JAR file I analyzed, I would add it to the Java classpath and pass its path

to the main method of CKJM-extended. If all required dependencies are present in the Java classpath, a textual output is generated which I saved. However, if all dependencies are not present in the Java classpath, an exception is thrown, and the next JAR file is analyzed. Table 2.5 shows the results of this analysis.

120,572	JAR files which could be analyzed by CKJM
106,221	JAR files which could not be analyzed by CKJM

Table 2.5 – CKJM analysis results.

In my previous attempts, I used Java 8 (OpenJDK 8). However, I was experiencing errors like 'org.apache.bcel.classfile.ClassFormatException: Invalid byte tag in constant pool'. This may be because some of the JAR files in the downloaded JAR repository had been compiled with versions of Java greater than Java 8 [25] [31]. To increase the number of JAR files that could be analyzed, I also tried adding all the JAR files in the downloaded JAR repository to the Java classpath. However, this resulted in 'jdk.internal.util.jar.InvalidJarIndexError: Invalid index' error. This error is thrown when the INDEX.LIST file included with a JAR file in the classpath points to missing dependencies [32]. Most likely some of the JAR files have an invalid INDEX.LIST file which causes this error.

2.2.2 JDepend

JDepend is another tool that can calculate size and software package metrics by traversing through Java class file directories in a package [16]. Table 2.6 shows the metrics calculated for every package.

Name of the metric	Metric description
Total Classes (NC)	Number of classes and interfaces in a package.
Concrete Classes (NCC)	Number of concrete classes in a package.
Abstract Classes (NAC)	Number of abstract classes and interfaces in a package.
Afferent Couplings (Ca)	Number of other packages that depend upon classes within a given package.
Efferent Couplings (Ce)	Number of other packages that the classes in a package depend upon.
Abstractness (A)	Ratio of the number of abstract classes and interfaces in a package to the total number of classes in that package.
Instability (I)	$I = \frac{Ce}{Ce + Ca}$

	Where I = Instability Ce = Efferent couplings Ca = Afferent couplings
Distance from Main Sequence (DMM)	Perpendicular distance of a package from the line $A + I = 1$ Where A = Abstractness I = Instability

Table 2.6 – JDepend field descriptions.

I also ran JDepend for every JAR file in the downloaded JAR repository using the 'executor' Python script mentioned previously. JDepend can scan through multiple packages in a JAR file. For every JAR file, the tool produces a single output file containing information about every single package in that JAR file. For every package in a given JAR file, in addition to the software package metrics [33], JDepend also shows the fully qualified names of all abstract and concrete classes in that package, the names of all packages on which the given package depends upon and the names of all packages which use the given package. Figure 2.4 shows a sample output in XML format. JDepend can be configured to display graphical, textual or XML output. For my purposes, I chose the XML format as it could be reliably converted to JSON format using the 'xmltodict' Python library [34]. Once I had completed generating XML files for every JAR file in the downloaded JAR repository, I ran the 'database' script to save the metrics. The 'database' script used the 'xmltodict' library to parse one XML file at a time and store the JAR file names, their constituent package names and the metric values for every package in a MariaDB database. I used Java 15 (OpenJDK 15) and JDepend release 2.10 in my analysis. Table 2.7 shows the results of this analysis.

226,781	JAR files which could be analyzed by JDepend.
12	JAR files which could not be analyzed by JDepend.

Table 2.7 – JDepend analysis statistics.

```xml
<?xml version="1.0"?>
<JDepend>
    <Packages>

        <Package name="com.amazonaws.services.lambda.runtime">
            <Stats>
                <TotalClasses>10</TotalClasses>
                <ConcreteClasses>3</ConcreteClasses>
                <AbstractClasses>7</AbstractClasses>
                <Ca>0</Ca>
                <Ce>3</Ce>
                <A>0.7</A>
                <I>1</I>
                <D>0.7</D>
                <V>1</V>
            </Stats>

            <AbstractClasses>
                <Class sourceFile="Client.java">
                    com.amazonaws.services.lambda.runtime.Client
                </Class>
                <Class sourceFile="ClientContext.java">
                    com.amazonaws.services.lambda.runtime.ClientContext
                </Class>
                <Class sourceFile="CognitoIdentity.java">
                    com.amazonaws.services.lambda.runtime.CognitoIdentity
                </Class>
                <Class sourceFile="Context.java">
                    com.amazonaws.services.lambda.runtime.Context
                </Class>
                <Class sourceFile="LambdaLogger.java">
                    com.amazonaws.services.lambda.runtime.LambdaLogger
                </Class>
                <Class sourceFile="RequestHandler.java">
                    com.amazonaws.services.lambda.runtime.RequestHandler
                </Class>
                <Class sourceFile="RequestStreamHandler.java">
                    com.amazonaws.services.lambda.runtime.RequestStreamHandler
                </Class>
            </AbstractClasses>

            <ConcreteClasses>
                <Class sourceFile="LambdaRuntime.java">
                    com.amazonaws.services.lambda.runtime.LambdaRuntime
                </Class>
                <Class sourceFile="LambdaRuntime.java">
                    com.amazonaws.services.lambda.runtime.LambdaRuntime$1
                </Class>
                <Class sourceFile="LambdaRuntimeInternal.java">
                    com.amazonaws.services.lambda.runtime.LambdaRuntimeInternal
                </Class>
```

Figure 2.4 – JDepend output.

2.2.3 MariaDB database schema

I used a MariaDB database to save the metrics calculated by CKJM-extended and JDepend. Figure 2.5 shows the schema of this database. For every JAR file that I was able to analyze with CKJM-extended, I was able to extract all constituent class names and their corresponding design metrics. Similarly, for every JAR file that I was able to analyze with JDepend, I was able to extract all constituent package names and their corresponding design metrics. Thus, I had both package-level and class-level metrics.

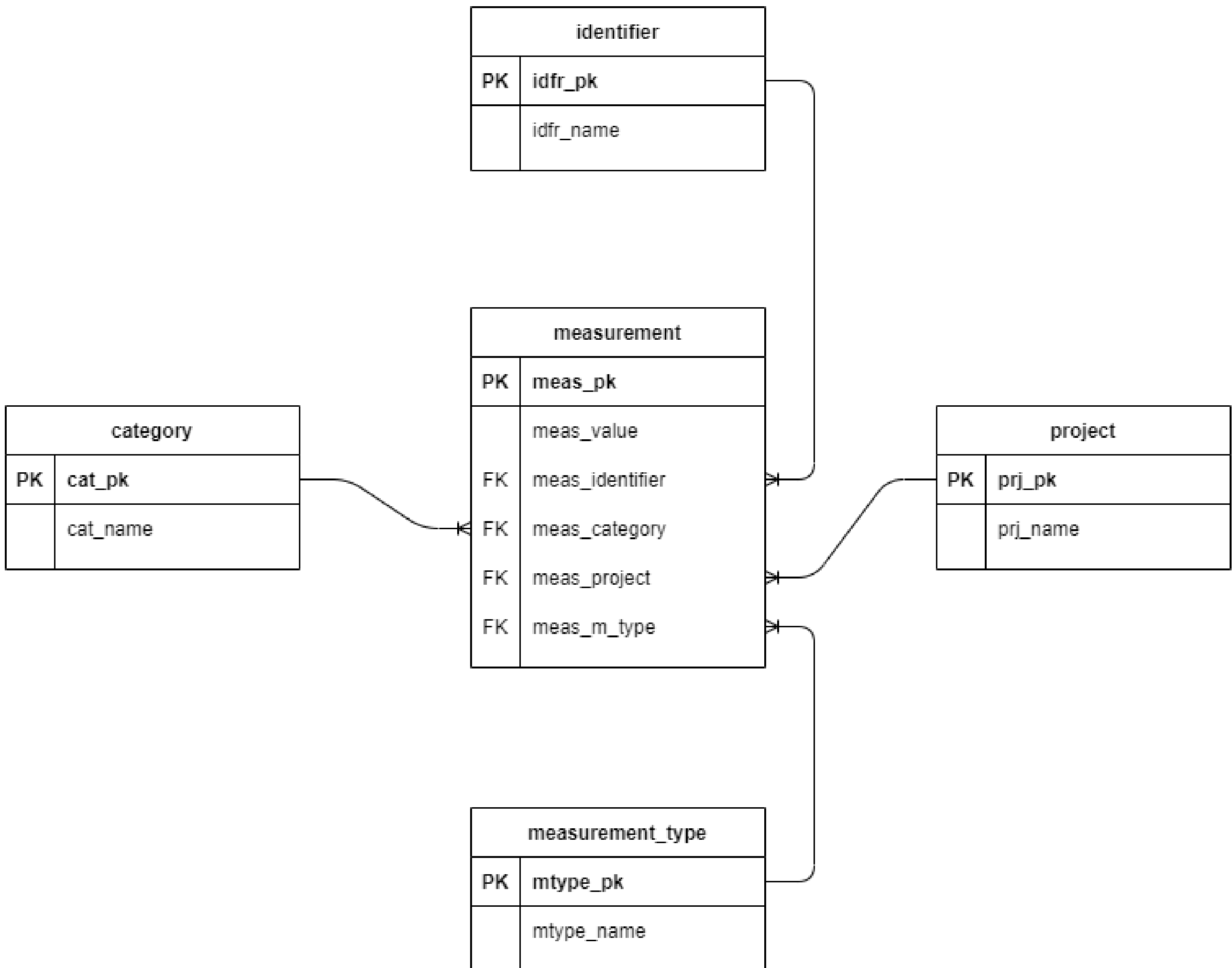


Figure 2.5 – Database schema.

To effectively analyze the metrics I had extracted, I stored every metric measurement in the 'measurement' table and associated it with four parameters stored in the other tables. The 'measurement' table stores the numerical metric value calculated by the tools e.g. 5, 80 etc. Every entry in the 'measurement' table is linked via foreign key constraints to the other

four tables. The ‘measurement_type’ table gives the name of the metric associated with a given measurement. Thus, it could be ‘NumberOfClasses_jdepend’ which tells us that the measurement is a value of the ‘Number of Classes’ metric calculated by JDepend. The ‘project’ table gives the name of the JAR file associated with a given measurement. Thus, it could be ‘abbot-1.4.0’ which tells us that the measurement was taken from the ‘abbot-1.4.0.jar’ file. The ‘identifier’ table gives the fully qualified name of the class/package associated with a given measurement. Thus, it could be ‘abbot.finder’ which tells us that the measurement is some metric of the ‘abbot.finder’ package. The ‘category’ table tells whether a given measurement is at the ‘package’ level or at the ‘class’ level. Table 2.8 summarizes the significance of each table.

Table name	Purpose of table
Measurement	The numerical value of a measured metric.
Measurement_type	The type of metric measured.
Project	The name of JAR file from which metric was measured.
Identifier	The name of class/package for which metric was measured.
Category	The level of metric: class level/package level.

Table 2.8 – Database table significance.

In the following chapters I analyze my findings in detail.

Chapter 3

Analysis of the Dataset

3.1 Approach to analysis

After analyzing every JAR file in the downloaded JAR repository with CKJM-extended, I was able to extract measurements for 18 class-level metrics for 10,608,920 classes from 120,572 JAR files. Similarly, with JDepend I was able to extract measurements for 8 package-level metrics for 2,107,577 packages from 226,781 JAR files. I extracted all measurements for each of these metrics into separate CSV files and plotted them with Matplotlib and Pandas to get the distribution of every metric. Along with the distribution plots I also calculated the minimum, maximum, 25th, 50th, 75th percentiles, mean and standard deviation for the measurement values of each of these metrics.

From the distribution plots for class-level metrics WMC, NPM, RFC, AMC, LCOM, CBO, Ca, Ce, DIT, NOC, IC, CBM, LOC and MOA, I observed that most of the 10,608,920 classes had metric values below a certain number (that varies by every metric) and a very few thousand classes had metric values which were much greater than this number. I consider the top 10,000 (0.0943%) classes as the outlier classes for these metrics. Using Cochran's formula for maximum variability, 50% confidence interval and $\pm$ 20% precision, I randomly sampled 12 of these outlier classes for every metric. For the class-level metrics LCOM3, CAM, MFA, DAM which are ratios between 0 and 1 (0 and 2 in case of LCOM3), I randomly sampled 5 classes from all 10,608,920 classes with metric values in different intervals.

Similarly, from the distribution plots for package-level metrics like number of classes, Ca and Ce, I observed that most of the 2,107,577 packages had metric values below a certain number and a very few hundred packages had metric values which were much greater than this number. I consider the top 1,000 (0.0474%) packages as the outlier packages for these

metrics. I randomly sampled 12 of these outlier packages for every metric. For the package-level metrics A, I and DMM, I randomly sampled 5 packages from all 2,107,577 packages with metric values in different intervals.

While gathering the outliers for every metric, I observed that many of the outlier JARs were outliers in terms of more than one metric. This suggested that there was some correlation between the metrics. To confirm this, I plotted the correlation matrix for the class-level and package-level metrics for the union of outliers for every metric. Figure 3.1 and figure 3.20 show these plots for package-level and class-level metrics, respectively. The correlation matrices indicate the value of Pearson's correlation coefficient for every metric pair. This value ranges from -1 to +1. A value of +1 means that a linear equation perfectly describes the relation between metric X and metric Y, where Y increases if X is increased. A value of -1 means that a linear equation perfectly describes the relation between metric X and metric Y, where Y decreases if X is increased. A value of 0 means that there is no linear relation between metric X and metric Y. For my purposes, I consider two metrics to be correlated if the absolute value of their Pearson's correlation coefficient is greater than 0.3.

From the correlation matrices, I observed regions where there existed low to very high correlation between metric pairs. To further explore these correlations, I sampled classes/packages which were outliers for multiple metrics and examined their source code. In the following sections, I present my findings for every group of correlated metrics. For every metric, I also present one randomly sampled outlier class/package with details of its metric values in the appendix.

3.2 Class level analysis

Figure 3.1 shows the Pearson's correlation coefficient matrix for the 18 class-level metrics. There are seven distinct regions of correlated metric pairs. Metrics LOC, WMC, NPM, RFC and LCOM have a low to high correlation. Metrics Ce, CBO, Ca and NOC have a low to high correlation. Metrics DAM and LCOM3 have a moderate negative correlation. Metrics CAM and RFC have a slight negative correlation. Metrics DIT, MFA, IC and CBM have low to high

correlation. Metrics AMC and MOA do not show any correlation with any other metric. In the following sections, I explore each of these metric groups and present my findings.

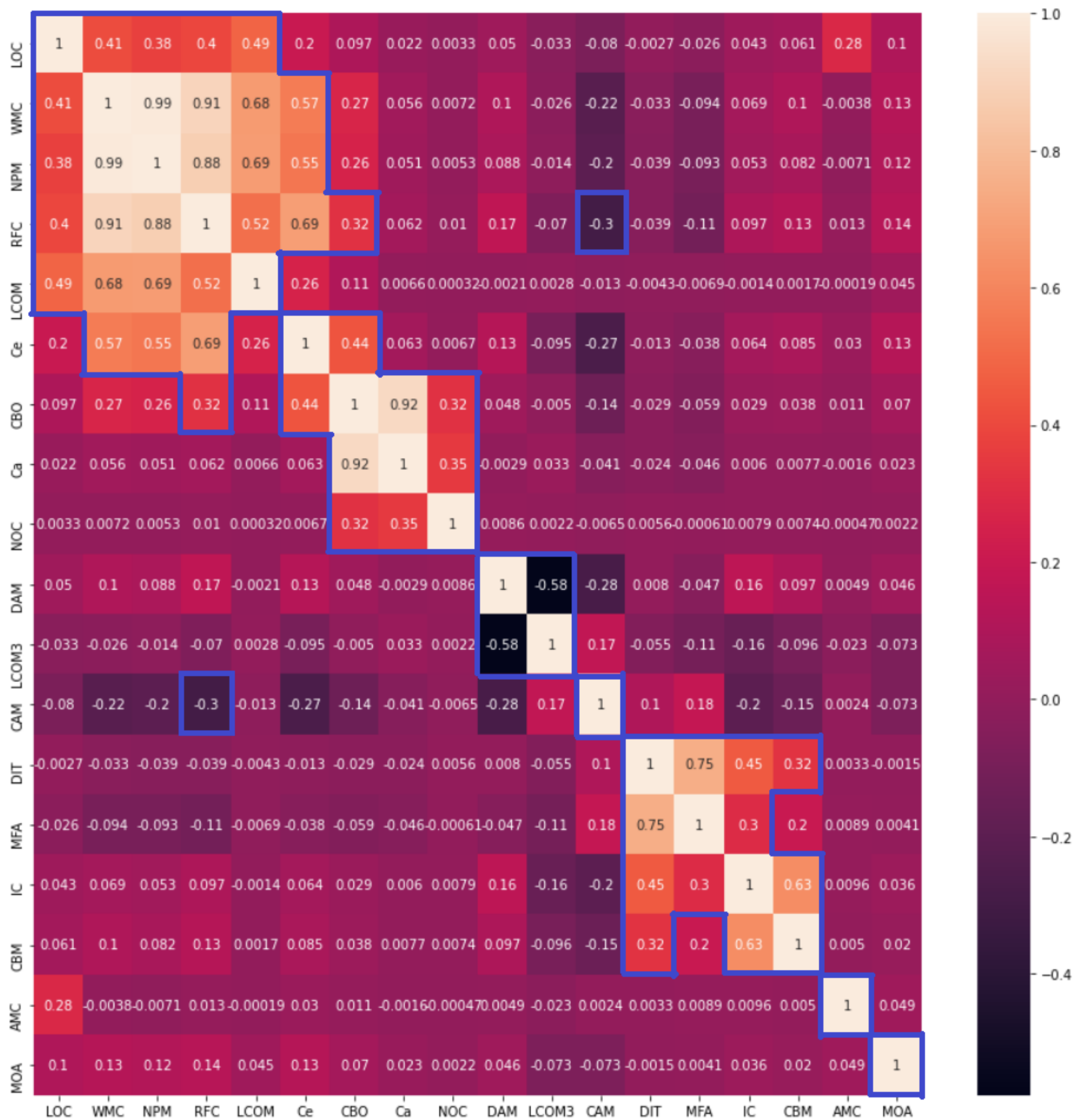


Figure 3.1 – Correlation matrix for class level metrics.

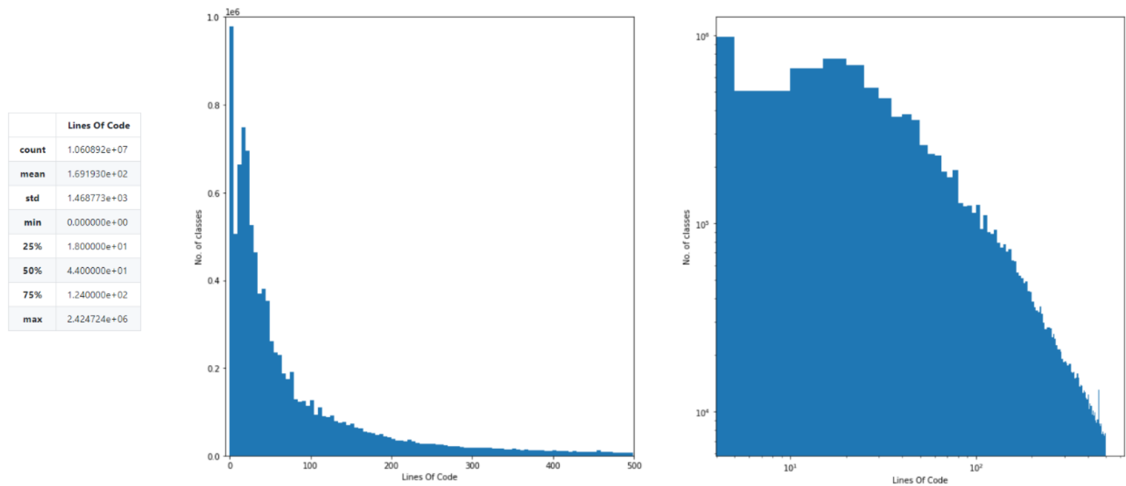
3.2.1 Size metrics (LOC, WMC, NPM, RFC and LCOM)

LOC gives the number of lines of code in a class. WMC gives the number of methods in a class. NPM gives the number of public methods in a class. RFC gives the number of methods that could be called by an object of a class in response to a message received by that object. LCOM gives the difference between the number of method pairs which do not share fields

of a class and the number of method pairs which share fields of a class. The distribution plots for these metrics in figure 3.2 to figure 3.6 show that 75% classes have up to 124 lines of code, 10 methods, 8 public methods. 75% classes can call up to 22 methods in response to a message and have an LCOM up to 22. The distributions for these metrics also suggest that most classes have small values for these metrics, but the few thousand outliers have much larger values. I sampled outliers based on the LOC metric. The 10,000 outliers have more than 8,867 lines of code, with the maximum number being 2,424,724 lines of code. The 10,000 outlier classes contribute 205,868,829 lines of code out of the 1,794,955,210 lines of code contributed by all classes. Thus 0.0943% classes contribute 11.4693% of the total lines of code.

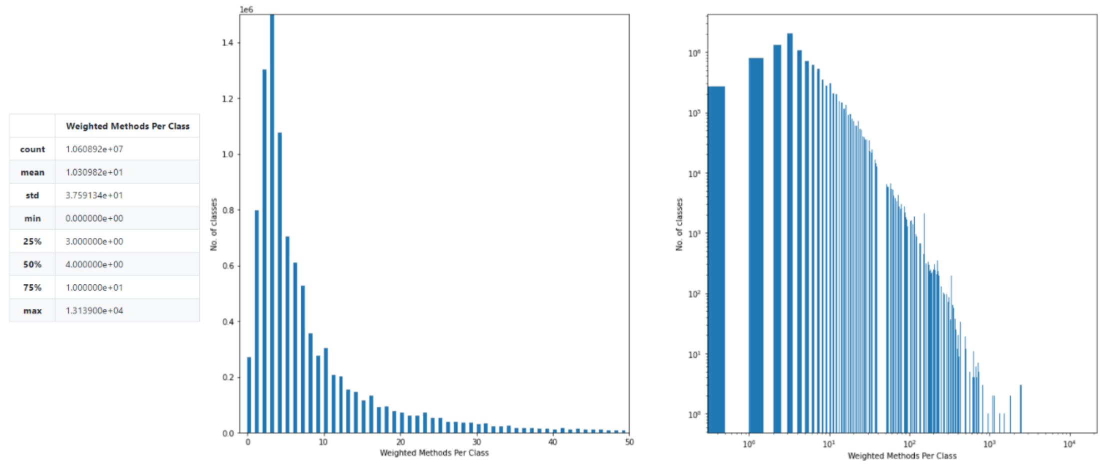
The sampled outliers included JDBC drivers, Scala classes, Scala libraries for asynchronous and concurrent programming, libraries to serialize/de-serialize JSON data to/from Java objects and APIs providing 3D graphics to applications written in Java. The database drivers had multiple autogenerated methods for different database operations. The Scala classes comprised of many static and synthetic methods generated by the compiler. Some other Scala classes had autogenerated methods that performed mathematical operations on collections of data. The serialization/de-serialization libraries had many autogenerated methods which manipulated the several elements of the JSON schemas they worked with. The 3D graphics APIs had several autogenerated methods to work with different graphical elements.

A common observation in the outliers for these metrics is that all of them had autogenerated or synthetic methods. The 3D graphics API was a wrapper library around code written in non-object-oriented languages. The database driver and Scala classes too were entirely made up of autogenerated or synthetic methods. Because of the large number of methods, any object of these classes can call many methods in response to a message. In addition to this, the moderate positive correlation with the CBO and Ce metrics matches with the observation that many of these autogenerated methods have method parameters of several other class types. Since the methods of these classes are autogenerated, they do not represent how code would normally be written using an object-oriented programming language and should be excluded from datasets for training machine learning models.



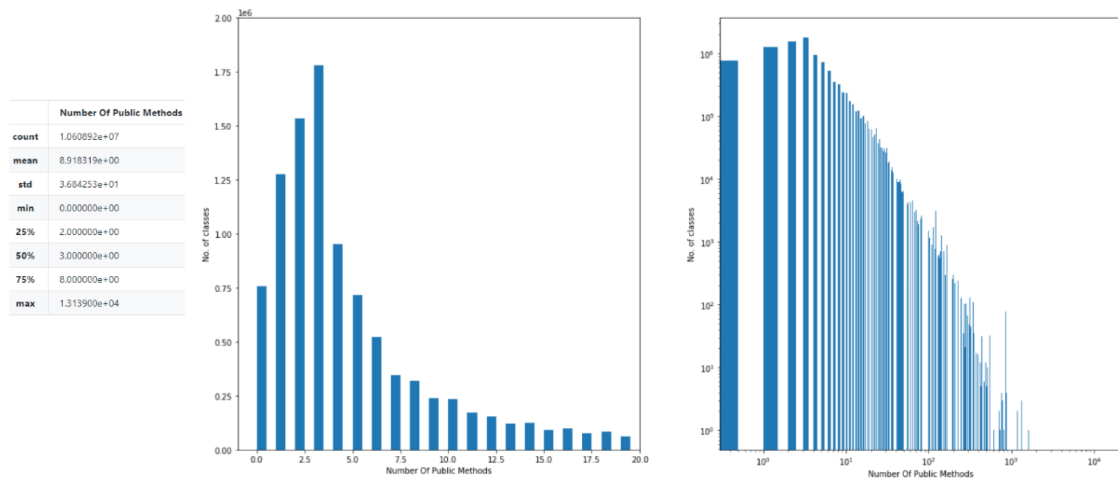
	Lines Of Code
count	1.060892e+07
mean	1.691930e+02
std	1.468773e+03
min	0.000000e+00
25%	1.800000e+01
50%	4.400000e+01
75%	1.240000e+02
max	2.424724e+06

Figure 3.2 – Lines of Code



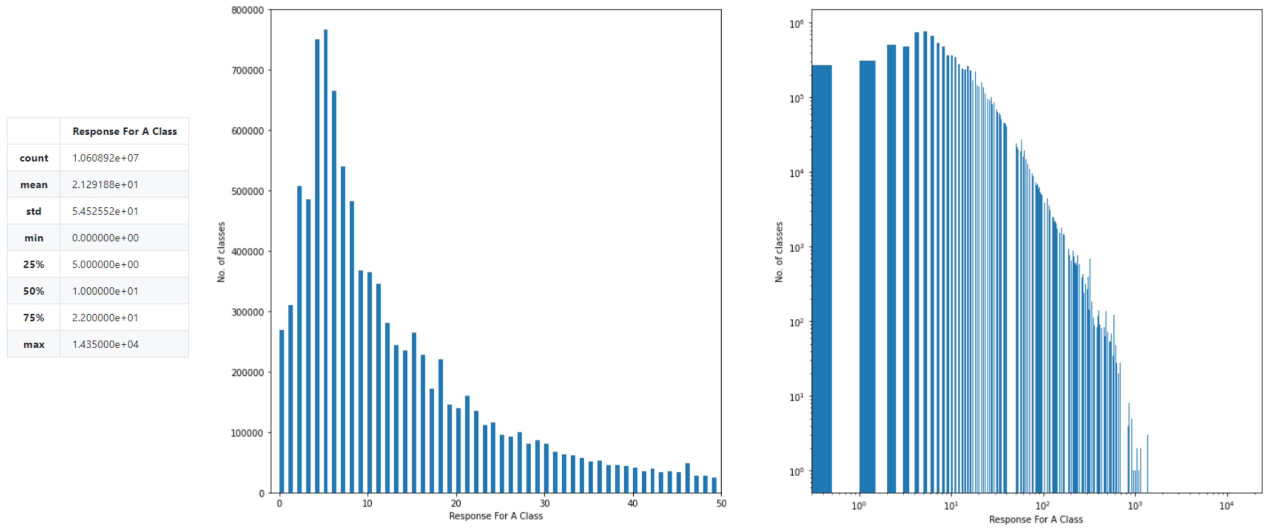
	Weighted Methods Per Class
count	1.060892e+07
mean	1.030982e+01
std	3.759134e+01
min	0.000000e+00
25%	3.000000e+00
50%	4.000000e+00
75%	1.000000e+01
max	1.313900e+04

Figure 3.3 – Weighted Methods per Class



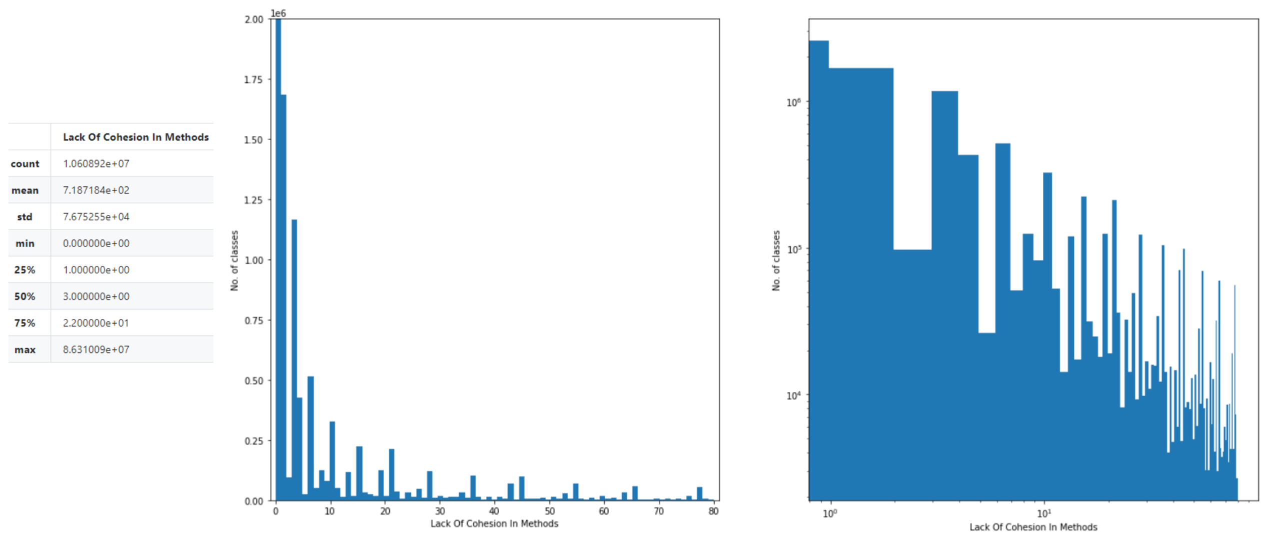
	Number Of Public Methods
count	1.060892e+07
mean	8.918319e+00
std	3.684253e+01
min	0.000000e+00
25%	2.000000e+00
50%	3.000000e+00
75%	8.000000e+00
max	1.313900e+04

Figure 3.4 – Number of Public Methods



	Response For A Class
count	1.060892e+07
mean	2.129188e+01
std	5.452552e+01
min	0.000000e+00
25%	5.000000e+00
50%	1.000000e+01
75%	2.200000e+01
max	1.435000e+04

Figure 3.5 – Response for a Class



	Lack Of Cohesion In Methods
count	1.060892e+07
mean	7.187184e+02
std	7.675255e+04
min	0.000000e+00
25%	1.000000e+00
50%	3.000000e+00
75%	2.200000e+01
max	8.631009e+07

Figure 3.6 – Lack of Cohesion in Methods

3.2.2 Coupling metrics (Ce, CBO, Ca and NOC)

Ce gives the number of classes used by the class for which it is measured. CBO gives the number of classes coupled to a class through inheritance, field access, method calls etc. Ca gives the number of classes which use the class for which it is measured. NOC gives the number of immediate child classes of a class. The distribution plots for these metrics in figure 3.7 to figure 3.10 show that 75% classes use up to 8 other classes, are coupled with

up to 10 classes, are used by only one other class and have up to one child class. The distributions for these metrics are like those in the previous set where most classes have a small metric value, but the few thousand outliers have significantly larger values. I sampled outliers based on the CBO metric. The 10,000 outlier classes were coupled with more than 299 classes with the maximum coupling being 15,560. The 10,000 outlier classes contribute 6,086,613 class couplings out of the 105,790,665 class couplings contributed by all classes. Thus 0.0943% classes contribute 5.7534% of the total couplings.

The sampled outliers included interfaces in Java clients for various APIs, abstract classes or interfaces taken from message parsing APIs and APIs for communication protocols, interfaces taken from Java compile-time generators and decorators and classes from a functional programming language that compiles to Java and runs on the JVM. The interfaces in the Java clients for APIs had many nested interfaces along with several methods which would return objects implementing these nested interfaces. The interfaces from the Java compile-time generators and decorators had a single abstract method which was implemented by several other classes in the same project.

A common observation in the outliers for these metrics is that all of them were either interfaces or abstract classes. In object-oriented programming, interfaces and abstract classes consist of abstract methods which define a set of behaviors that are expected from any class that implements or extends them. This general set of behaviors can be specialized to suit the needs of the implementing/extending class. Consider for example a 'Vehicle' interface with a 'drive' method. Considering the large range of vehicles that can be driven, it is likely that such an interface would be implemented by many concrete classes. Thus, it is not surprising to find interfaces and abstract classes being used by many classes. All the outliers were used by many other classes in their projects either as method parameter types or method return types. Most of these outliers were also implemented/extended by several other classes in their project. This resulted in the large number of couplings with other classes. The positive correlation between coupling metrics and number of direct descendants indicates the need to add more levels to the hierarchy tree and split the direct descendants among the newly added subclasses/sub-interfaces. This will ensure loose coupling and code reuse at the same time.

	Efferent Couplings
count	1.060892e+07
mean	6.548784e+00
std	1.213528e+01
min	0.000000e+00
25%	2.000000e+00
50%	4.000000e+00
75%	8.000000e+00
max	4.796000e+03

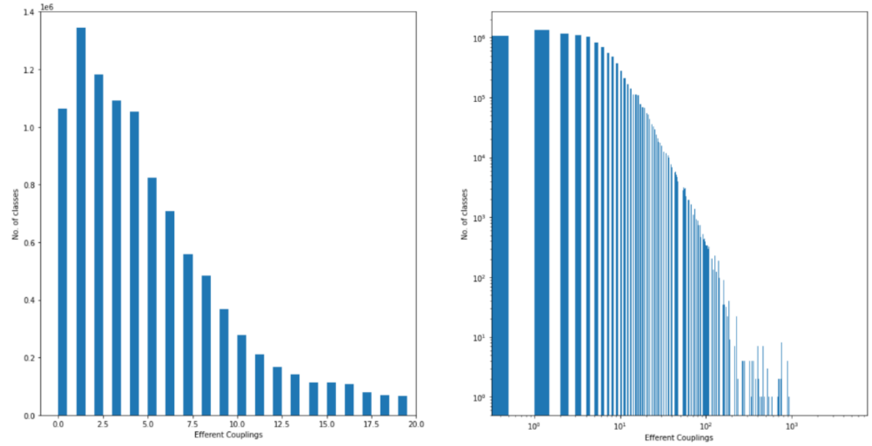


Figure 3.7 – Efferent Couplings

	Coupling Between Object Classes
count	1.060892e+07
mean	9.971856e+00
std	2.968492e+01
min	0.000000e+00
25%	3.000000e+00
50%	6.000000e+00
75%	1.000000e+01
max	1.556000e+04

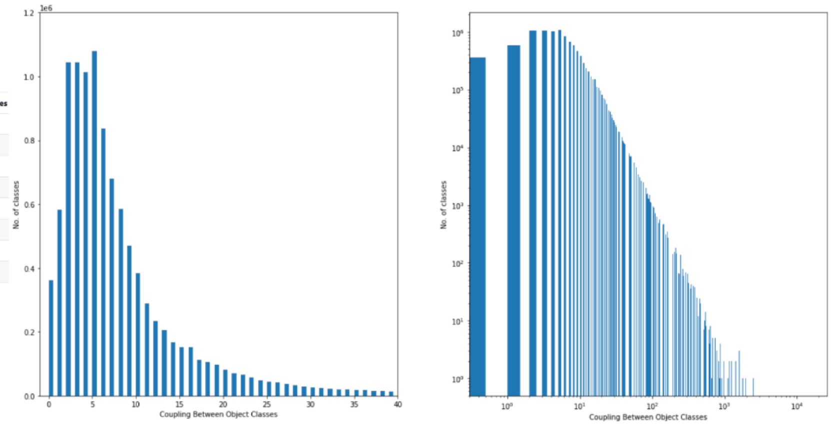


Figure 3.8 – Coupling Between Object Classes

	Afferent Couplings
count	1.060892e+07
mean	4.014364e+00
std	2.698834e+01
min	0.000000e+00
25%	1.000000e+00
50%	1.000000e+00
75%	3.000000e+00
max	1.555700e+04

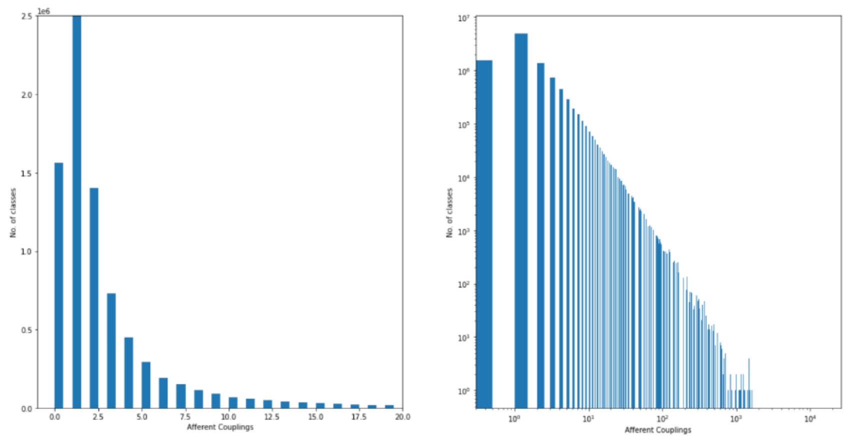
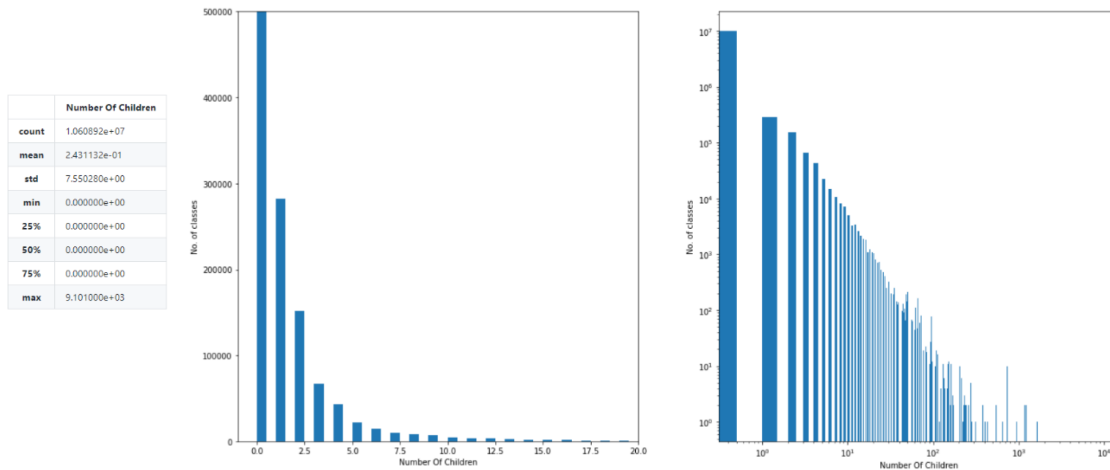


Figure 3.9 – Afferent Couplings



	Number Of Children
count	1.060892e+07
mean	2.431132e-01
std	7.550280e+00
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	0.000000e+00
max	9.101000e+03

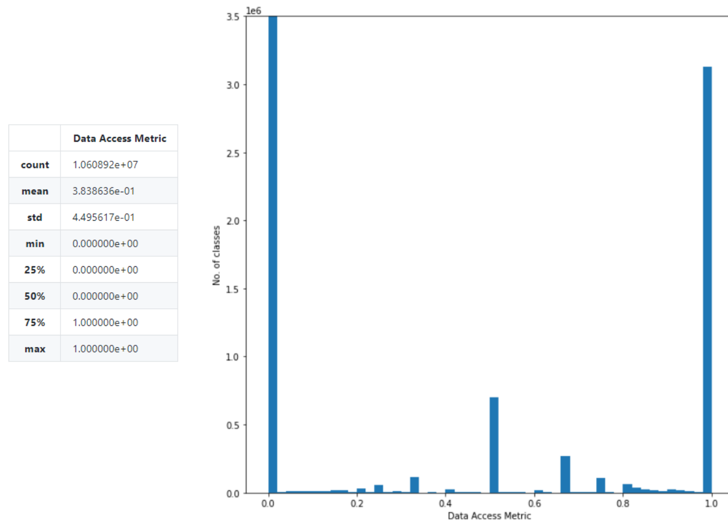
Figure 3.10 – Number of Children

3.2.3 Encapsulation and Cohesion metrics (DAM and LCOM3)

DAM gives the ratio of private class attributes to total class attributes. LCOM3 gives the degree of cohesiveness of the methods in a class. LCOM3 varies from 0 to 2. LCOM3 = 0 indicates that every class method accesses every class attribute. This is the highest degree of cohesiveness. Higher values of LCOM3 result when some or all class methods do not access some or all class attributes. The distribution plots for these metrics in figure 3.11 and 3.12 show that only 50% classes have DAM greater than zero. 25% of the classes have LCOM3 less than 0.62, 25% classes have LCOM3 between 0.62 and 0.91 and the remaining 50% classes have LCOM3 greater than 0.91.

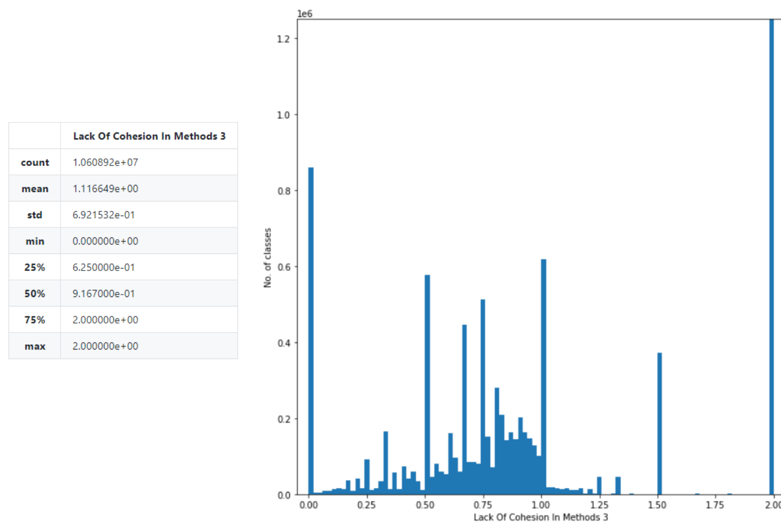
The correlation matrix for class-level metrics in figure 3.1 indicates a moderate negative correlation between DAM and LCOM3. This means that classes with a greater proportion of private attributes tend to have lower lack of cohesion among their methods. In other words, classes with a greater proportion of private attributes tend to have cohesive methods. I sampled five classes with LCOM3 = 0.0. The sampled classes were taken from a cryptography API for Java. A common observation in the sampled classes was that all had private attributes with methods that accessed most of them. As an example, class 'org.bouncycastle.asn1.smime.SMIMECapabilityVector' in project 'bcprov-jdk14-1.65' has a single private attribute of type 'org.bouncycastle.asn1.ASN1EncodableVector'. There are

three public methods which take objects of type ‘org.bouncycastle.asn1 .ASN1ObjectIdentifier’ and ‘org.bouncycastle.asn1.ASN1EncodableVector’ and add to the private attribute. The fourth public method returns the private attribute. Thus, the only attribute of this class is private and is accessed by all methods of that class. This results in the good scores for DAM and LCOM3. I also sampled five classes with $1.7 < LCOM3 < 1.8$. A common observation in these sampled classes was that all had static and final (constant) class attributes which were not accessed by any class methods.



	Data Access Metric
count	1.060892e+07
mean	3.838636e-01
std	4.495617e-01
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	1.000000e+00
max	1.000000e+00

Figure 3.11 – Data Access Metric



	Lack Of Cohesion In Methods 3
count	1.060892e+07
mean	1.116649e+00
std	6.921532e-01
min	0.000000e+00
25%	6.250000e-01
50%	9.167000e-01
75%	2.000000e+00
max	2.000000e+00

Figure 3.12 – Lack of Cohesion in Methods 3

3.2.4 Inheritance metrics (DIT, MFA, IC and CBM)

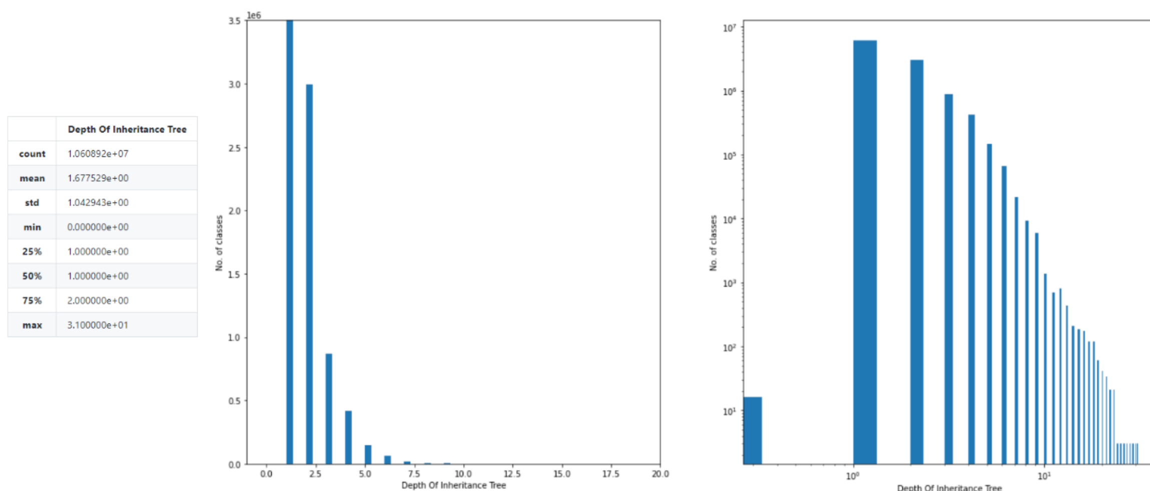
DIT gives the depth of a class from the top of the object hierarchy. MFA gives the ratio of inherited class methods to total methods accessible by member methods of that class (not considering the methods inherited from the root of the object hierarchy). IC gives the number of parent classes to which a class is coupled by method calls between inherited and new/redefined methods. CBM gives the number of new/redefined methods in a class with which all inherited methods are coupled by method calls. The distribution plots for these metrics in figure 3.13 to figure 3.16 show that 75% classes are up to 2 levels below the root of the object hierarchy. Most classes are not coupled with any parent class by method calls between inherited and new/redefined methods. Most classes do not have any new/redefined methods with which all inherited methods are coupled by method calls. Most classes have zero functional abstraction. 25% of the classes have an abstraction between 0 and 0.83, while 25% of the classes have an abstraction greater than 0.83.

I sampled outliers based on the IC metric. The 10,000 outlier classes are coupled with more than 4 parent classes with the maximum number being 17 parent classes. The 10,000 outlier classes contribute 50,555 inheritance couplings out of the 3,010,975 inheritance couplings contributed by all classes. Thus 0.0943% classes contribute 1.679% of the total inheritance couplings. The sampled outliers included classes from libraries for parsing CSV files to Java POJOs, classes from APIs for Scala.js, classes from a Reactive Streams foundation for the JVM and toolkits for Java.

A common observation in the outliers for these metrics is that all of them had many new/redefined methods which not only reused code inherited from methods of parent classes, but also added their own code on top of it. In object-oriented programming inheritance allows greater reuse of code. It involves parent classes with methods that provide generic behavior. These parent classes can be extended by one or more child classes. The child classes not only inherit methods from the parent class but are also free to define their own new methods or redefine the inherited methods. All the outliers studied were many levels below the root of the object hierarchy. They had inherited methods from the many parent classes above them. This resulted in a greater proportion of inherited methods. In addition to the inherited methods, they also had many new/redefined methods

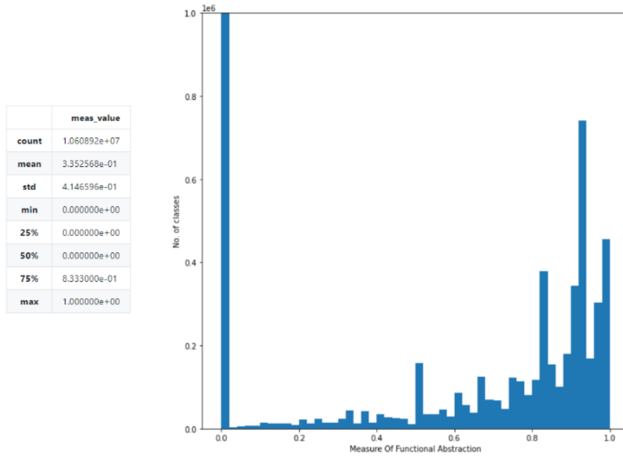
which were coupled with every inherited method by method calls. The need for these method calls can be explained by the fact that many of the new/redefined methods provide additional functionality on top of that provided by the parent classes. This requires invoking inherited methods from the new/redefined methods. As a result of these method calls, these classes were coupled with several parent classes.

One of the outliers was class ‘reactor.util.function.Tuple8’ in project ‘reactor-core-3.3.4.RELEASE’. This class has classes ‘Tuple7’, ‘Tuple6’, ‘Tuple5’, ‘Tuple4’, ‘Tuple3’, ‘Tuple2’ in that order above it in the hierarchy tree. Class ‘Tuple2’ implements the ‘Iterable’ and ‘Serializable’ interfaces. Every class in the hierarchy tree from ‘Tuple8’ to ‘Tuple2’ defines its own versions of the ‘equals’ and ‘hashCode’ methods. Class ‘Tuple8’ for example inherits these methods from class ‘Tuple7’. Class ‘Tuple8’ also redefines these methods. Inside the ‘Tuple8’ definition of these methods, method calls are made to the ‘Tuple7’ definition of these methods (using the ‘super’ keyword). As a result, class ‘Tuple8’ can not only reuse code form class ‘Tuple7’, but also add its own code to it. This leads to greater reuse of code and the inheritance coupling between classes ‘Tuple8’ and ‘Tuple7’. The positive correlation between DIT, IC and CBM metrics suggests that greater use of inheritance allows greater reuse of code.



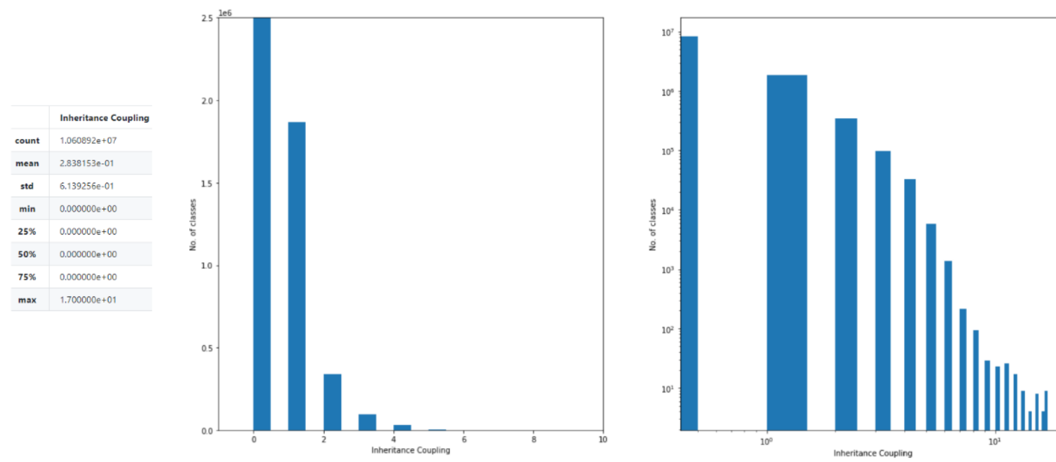
	Depth Of Inheritance Tree
count	1.060892e+07
mean	1.677529e+00
std	1.042943e+00
min	0.000000e+00
25%	1.000000e+00
50%	1.000000e+00
75%	2.000000e+00
max	3.100000e+01

Figure 3.13 – Depth of Inheritance Tree



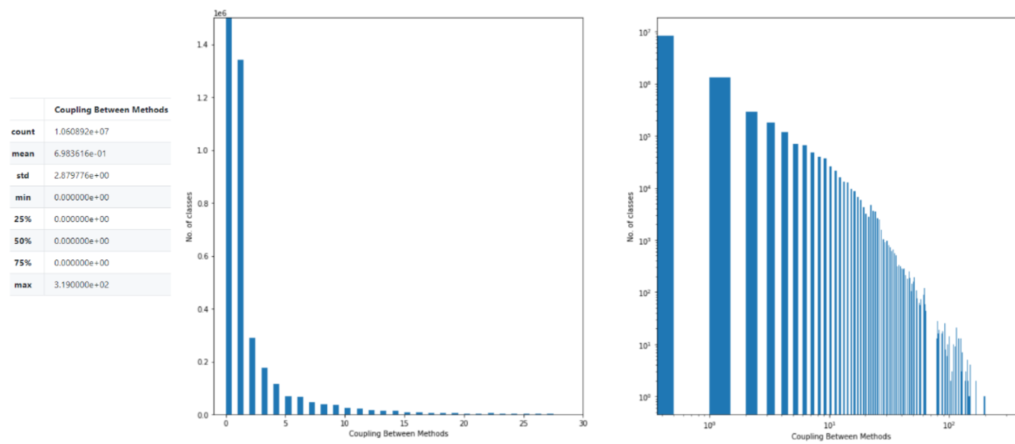
	meas_value
count	1.060892e+07
mean	3.352568e-01
std	4.146596e-01
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	8.333000e-01
max	1.000000e+00

Figure 3.14 – Measure of Functional Abstraction



	Inheritance Coupling
count	1.060892e+07
mean	2.838153e-01
std	6.139256e-01
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	0.000000e+00
max	1.700000e+01

Figure 3.15 – Inheritance Coupling



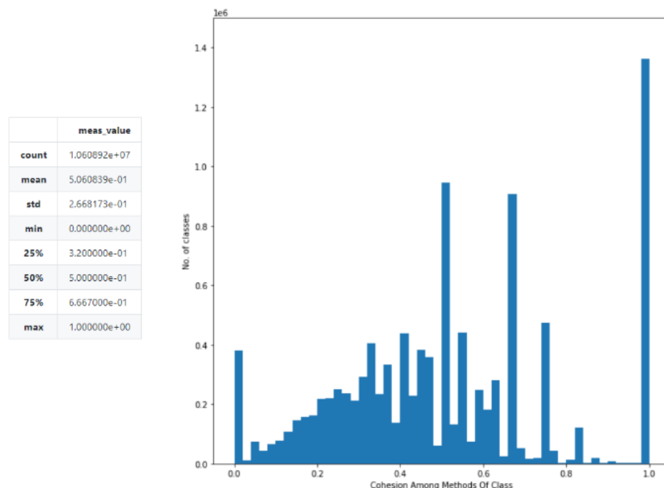
	Coupling Between Methods
count	1.060892e+07
mean	6.983616e-01
std	2.879776e+00
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	0.000000e+00
max	3.190000e+02

Figure 3.16 – Coupling Between Methods

3.2.5 Cohesion among methods of class (CAM)

CAM gives the degree of cohesiveness of the methods in a class. It varies from 0 to 1. CAM = 1 indicates that every class method uses every unique method parameter type in that class. This is the highest degree of cohesiveness. Lower values of CAM result when some or all class methods do not use every unique method parameter type. From figure 3.17, most classes have a score of one. 25% of the classes have a score of up to 0.32, 25% of the classes have a score between 0.32 and 0.5, 25% of the classes have a score between 0.5 and 0.67, and the remaining 25% classes have a score above 0.67.

Five classes with the CAM > 0.98 were sampled. The sampled classes can be classified into two categories. One category is the set of classes which do not have any methods other than the constructor. In these classes, the constructor takes a parameter of a certain type which is the only parameter type in the entire class. This results in a perfect score for cohesion. The other category is the set of classes which have a few methods all of which work with mostly the same parameter types. Classes in this category were taken from APIs that automate the mapping between XML documents and Java objects. A weak negative correlation was found between CAM and RFC. This could be explained by the observation that the sampled classes had specialized methods that did not call many other methods. Since this metric considers method parameter types rather than the class attributes accessed by methods, some classes had conflicting values of CAM and LCOM3.



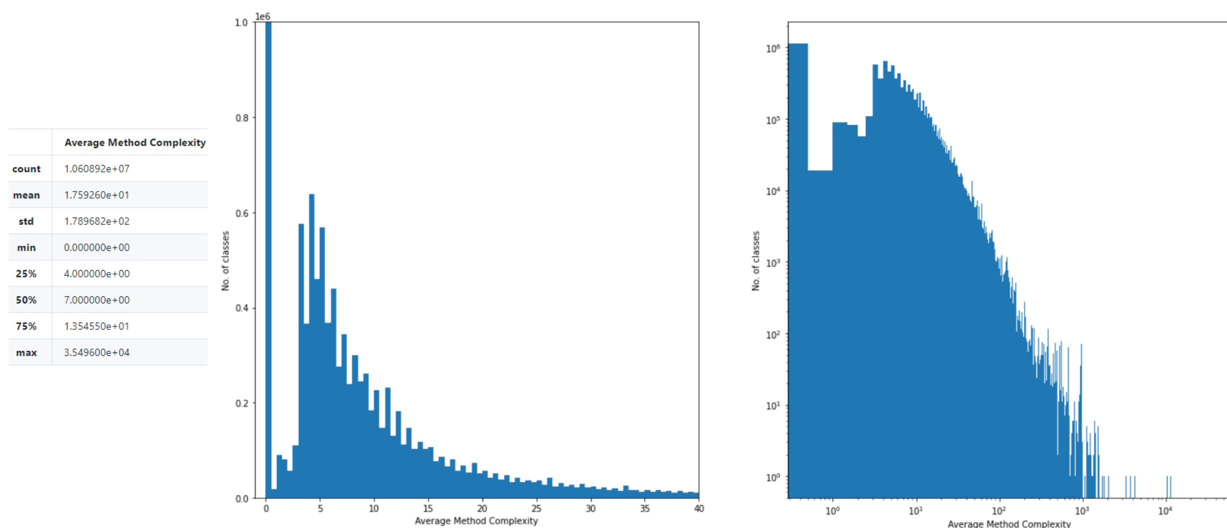
	meas_value
count	1.060892e+07
mean	5.060839e-01
std	2.668173e-01
min	0.000000e+00
25%	3.200000e-01
50%	5.000000e-01
75%	6.667000e-01
max	1.000000e+00

Figure 3.17 – Cohesion Among Methods of Class

3.2.6 Average method complexity (AMC)

AMC gives the average number of binary codes in all methods of a class. From figure 3.18, most classes have an average method complexity of 0, 3, 4, or 5 with 75% classes having an average method complexity less than 13.54. The 10,000 outlier classes have an average method complexity greater than 968.0 with the highest average method complexity being 35,496.0.

The sampled outliers included serializing/de-serializing libraries for AWS objects, Scala libraries for locales and time zones, Java libraries to generate PDFs, manage documents etc. The outlier classes from the libraries that serialized/de-serialized AWS objects had very large autogenerated constructors which added ‘mix-in’ annotations to several classes of the AWS SDK. Most of the outliers had zero or very few methods in addition to their constructors. The constructors were very large and had several lines of code which increased their average method complexity. No correlation was found between this and any other metric.



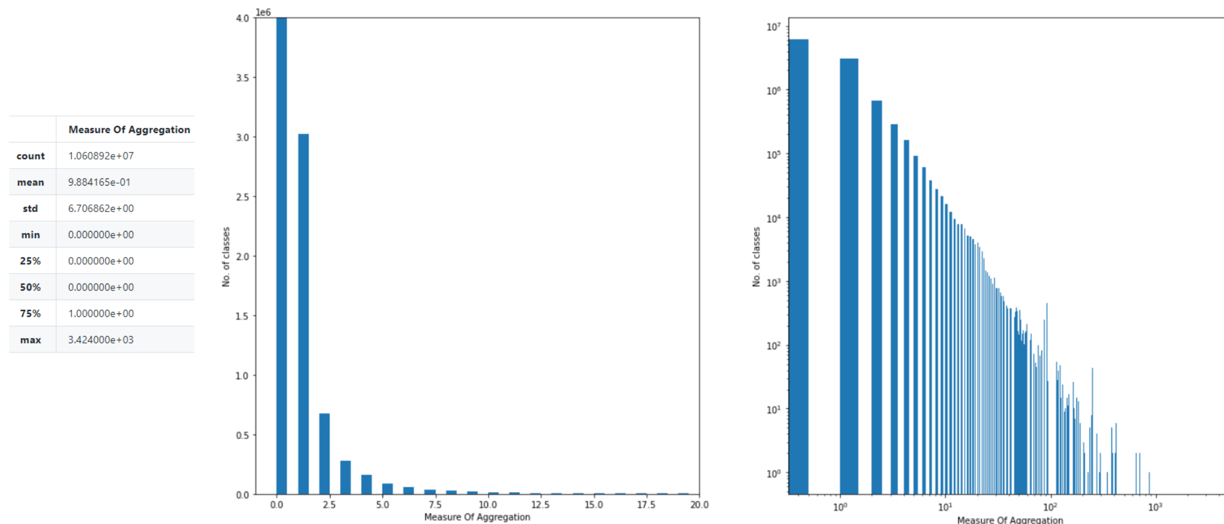
	Average Method Complexity
count	1.060892e+07
mean	1.759260e+01
std	1.789682e+02
min	0.000000e+00
25%	4.000000e+00
50%	7.000000e+00
75%	1.354550e+01
max	3.549600e+04

Figure 3.18 – Average Method Complexity

3.2.7 Measure of aggregation (MOA)

MOA gives the number of class attributes whose types are user defined classes. It gives the extent to which composition is used in a class. From figure 3.19, most classes have zero or one attribute whose type is a user defined class, with 75% classes having not more than one such attribute. The 10,000 outlier classes have more than 48 such attributes with the maximum number being 3,424 attributes.

The sampled outliers were mostly Scala classes and traits. One of the outliers was a Scala singleton object class 'com.vitorsvieira.iso.ISOCountrySubdivision' in project 'scala-iso_2.11-0.1.2' with 3,424 user defined class type attributes. This singleton object class extends multiple Scala traits like 'ISOCountrySubdivisionAfrica', 'ISOCountrySubdivisionAsia', 'ISOCountrySubdivisionEurope' etc. These Scala traits contain multiple fields of type 'ISOCountrySubdivision' for every country subdivision in the world. Considering the many country subdivisions in the world, the singleton object class 'ISOCountrySubdivision' gets a few thousand attributes of type 'ISOCountrySubdivision'. This results in a very high value for this metric. The other outliers also followed a similar pattern. No correlation was found between this and any other metric.



	Measure Of Aggregation
count	1.060892e+07
mean	9.884165e-01
std	6.706862e+00
min	0.000000e+00
25%	0.000000e+00
50%	0.000000e+00
75%	1.000000e+00
max	3.424000e+03

Figure 3.19 – Measure of Aggregation

3.3 Overall observations from class level analysis

The small number of outliers I sampled contribute a significant portion of the total lines of code, total number of methods, total number of couplings between classes and total number of inheritance couplings. If a machine learning model were to be trained by including these outliers, the outliers would have a significant influence over the performance of the model. Thus, one must be cautious when deciding which outliers to keep and which outliers to discard from the training set of the model. It is also important to consider the correlations between these metrics. I found that almost all outlier classes for the LOC, WMC, NPM, RFC and LCOM metrics had autogenerated code and did not really reflect how object-oriented programs could be written by a human programmer, some of them were even wrappers around code written in other languages. Considering this, these outliers can be safely discarded unless one wants to consider autogenerated code. The outliers for the Ce, CBO, Ca and NOC metrics were interfaces and abstract classes and are examples of implementing generalization-specialization in code. The outliers for the metrics DIT, MFA, IC and CBM show examples of increased code reusability due to a greater use of inheritance. The outliers for average method complexity were also autogenerated and can be discarded. The outliers for the MOA metric are examples of extreme use of composition in classes but do reflect a realistic use of composition.

3.4 Package level analysis

Figure 3.20 shows the Pearson's correlation coefficient matrix for the 8 package level metrics. There are three distinct regions of correlated metric pairs. Metrics NC, NCC and NAC have a moderate to high correlation. Metrics Ca, A, I and DMM have a moderately positive or negative correlation with each other. Metric Ce has negligible to zero correlation with the other metrics. In the following sections, I explore each of these metric groups and present my findings.

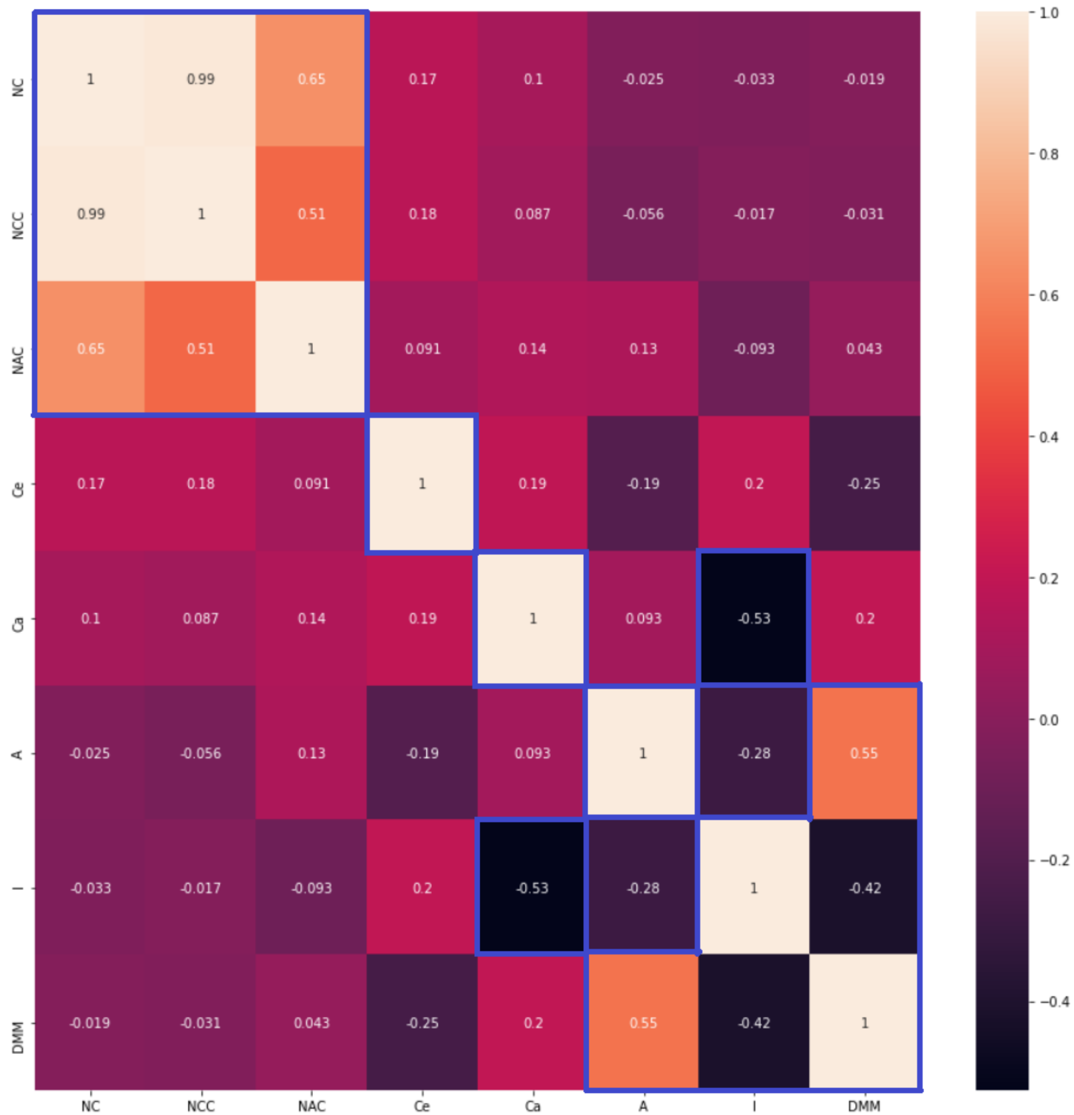


Figure 3.20 – Correlation matrix for package level metrics.

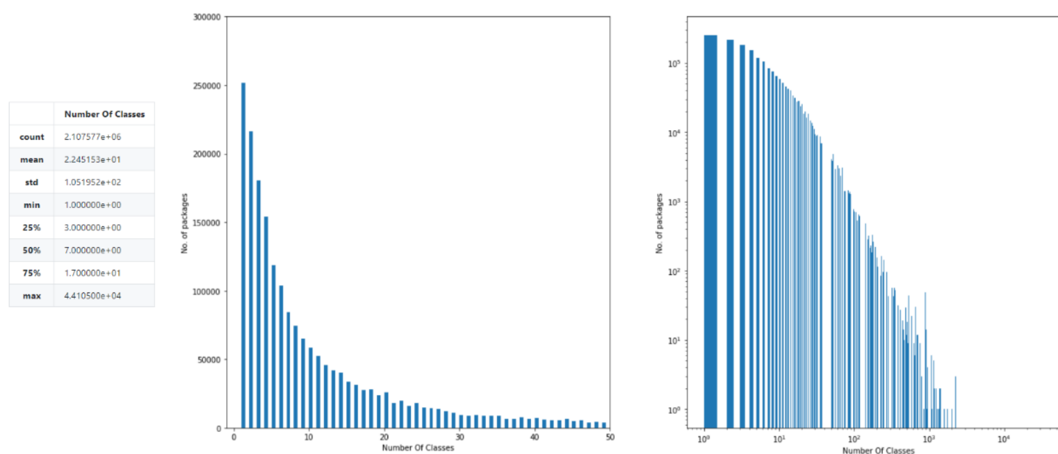
3.4.1 Size metrics (NC, NCC and NAC)

NC gives the total number of classes in a package. NCC gives the total number of concrete classes in a package. NOA gives the total number of abstract classes/interfaces in a package. The distribution plots for these metrics in figure 3.21 to figure 3.23 show that 75% packages have up to 17 total classes, 14 concrete classes, 3 abstract classes. The distributions for these metrics also suggest that most packages have very few concrete classes and zero or one abstract class, but the few hundred outlier packages have a significantly large number of classes. I sampled outliers based on the NC metric. The 1,000

outlier packages had more than 1,555 classes with the maximum number of classes being 44,105. The 1,000 outlier packages contribute 2,997,284 classes out of the 47,318,319 classes contributed by all packages. Thus 0.0474% packages contribute 6.3343% of the total classes.

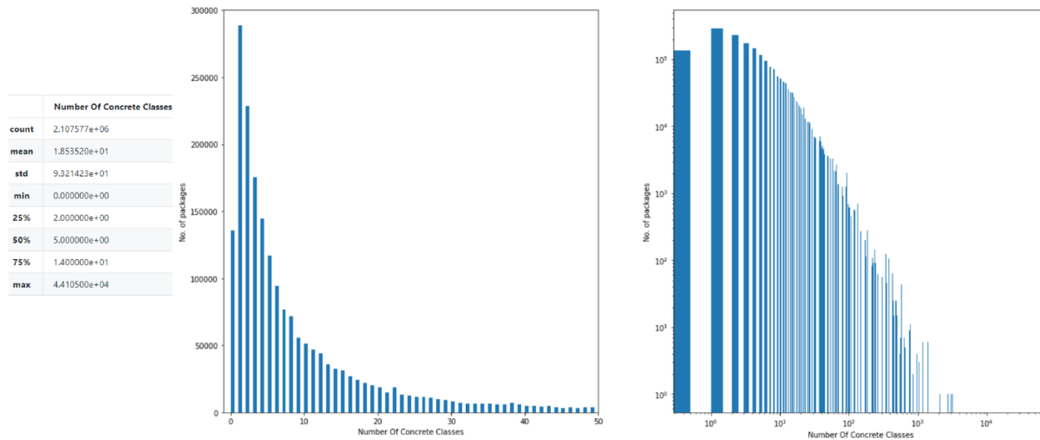
The sampled packages were mostly packages from projects like Apache Cassandra, Apache Zeppelin, Apache Spark, Google Guava Collections and Google Ads API client library.

A common observation in the outliers for these metrics is that all of them had a greater proportion of concrete classes. Because of the large number of classes in these packages, they were coupled with many other packages. The ratio of afferent and efferent couplings differed from package to package and no pattern could be found; however, the total number of couplings was higher than typical. As an example, package 'com.google.ads.googleads.v2.resources' in project 'google-ads-5.0.0' has 1,196 classes of which 1,039 are concrete and 157 are abstract. This package is used by three other packages including 'com.google.ads.googleads.v2.services' and 'com.google.ads.googleads.v2.utils'; and is used by 13 other packages including 'com.google.api', 'com.google.common.base', 'com.google.common.collect', 'com.google.ads.googleads.v2.common', 'com.google.ads.googleads.v2.enums', 'com.google.ads.googleads.v2.errors', 'java.lang', 'java.util' etc.



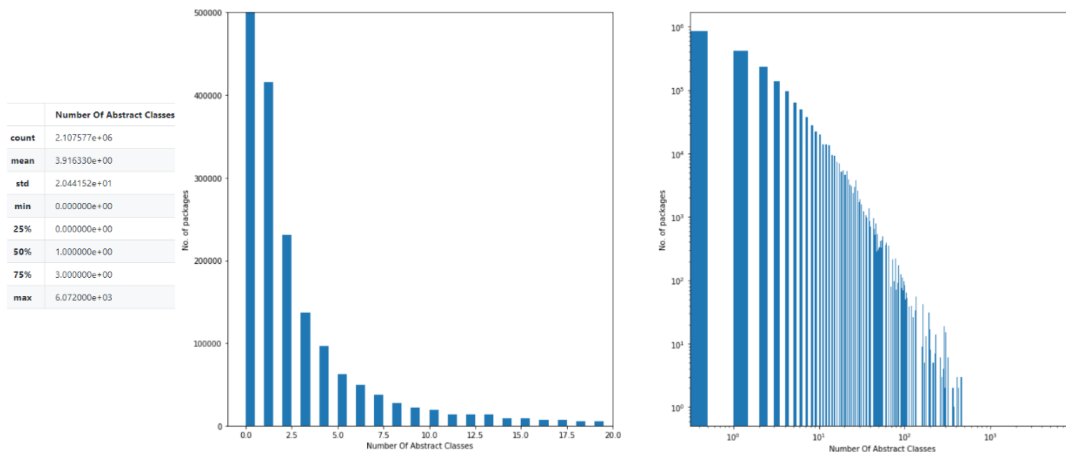
	Number Of Classes
count	2.107577e+06
mean	2.245153e+01
std	1.051952e+02
min	1.000000e+00
25%	3.000000e+00
50%	7.000000e+00
75%	1.700000e+01
max	4.410500e+04

Figure 3.21 – Number of Classes



	Number Of Concrete Classes
count	2.107577e+06
mean	1.853520e+01
std	9.321423e+01
min	0.000000e+00
25%	2.000000e+00
50%	5.000000e+00
75%	1.400000e+01
max	4.410500e+04

Figure 3.22 – Number of Concrete Classes



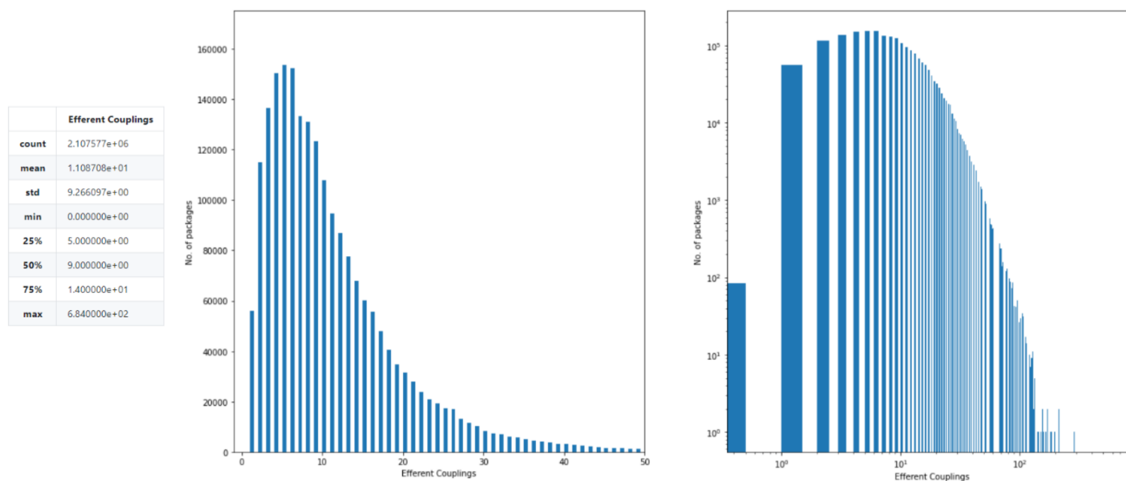
	Number Of Abstract Classes
count	2.107577e+06
mean	3.916330e+00
std	2.044152e+01
min	0.000000e+00
25%	0.000000e+00
50%	1.000000e+00
75%	3.000000e+00
max	6.072000e+03

Figure 3.23 – Number of Abstract Classes

3.4.2 Efferent couplings (Ce)

Ce gives the number of packages used by the package for which it is measured. From figure 3.24, most packages have 4 to 6 efferent couplings with 75% of the packages not having more than 14 efferent couplings. The 1,000 outlier packages had more than 90 efferent couplings with the maximum number of efferent couplings being 684. The 1,000 outlier packages contribute 113,468 efferent couplings out of the 23,366,883 efferent couplings contributed by all packages. Thus 0.0474% packages contribute 0.4856% of the total efferent couplings.

The sampled outliers were mostly specialized packages from projects like Eclipse GlassFish etc. A common observation in the outliers for these metrics is that most packages had classes that used the more general classes present in the core packages of their projects. As an example, package 'com.sun.enterprise.web' in project 'glassfish-embedded-all-5.1.0' uses 130 packages including 'org.glassfish.internal.api', 'javax.servlet.http', 'com.sun.enterprise.web.connector' etc. Each of these package dependencies holds core classes used by the various classes of 'com.sun.enterprise.web' package. The sampled outliers had more concrete classes than abstract classes. The correlation matrix in figure 3.20 does not show any correlation between Ce and any other metric even though there is a formula that relates I, Ca and Ce. This can be explained by the fact that the relationship between Ce and I is non-linear.



	Efferent Couplings
count	2.107577e+06
mean	1.108708e+01
std	9.266097e+00
min	0.000000e+00
25%	5.000000e+00
50%	9.000000e+00
75%	1.400000e+01
max	6.840000e+02

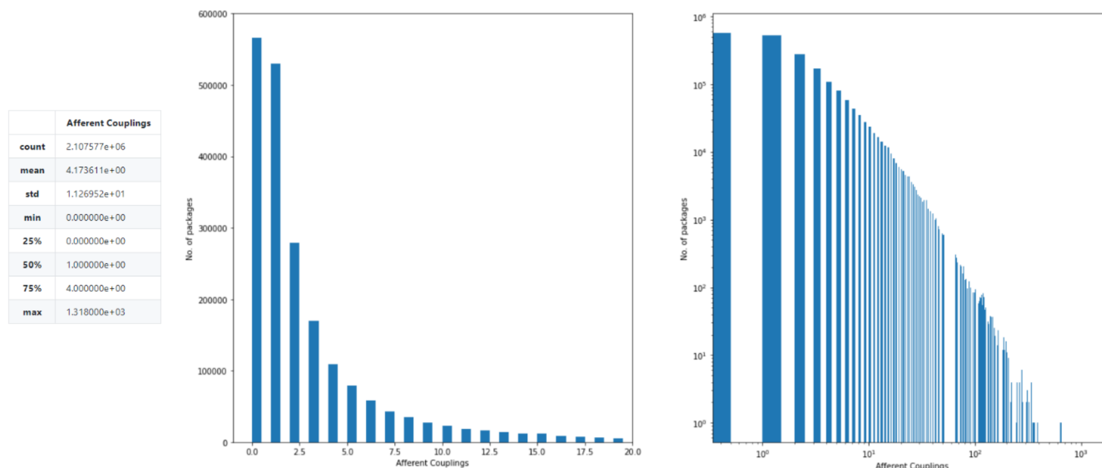
Figure 3.24 – Efferent Couplings

3.4.3 Popularity metrics (Ca, *A*, *I* and DMM)

Ca gives the number of packages which use the package for which it is measured. A gives the ratio of abstract classes and interfaces in a package to the total number of classes in that package. I gives the ratio of efferent couplings of a package to the total number of couplings of that package. DMM gives the perpendicular distance of a package from the line $A + I = 1$. It is desirable to have a DMM of 0 as it indicates the right balance between abstractness and instability.

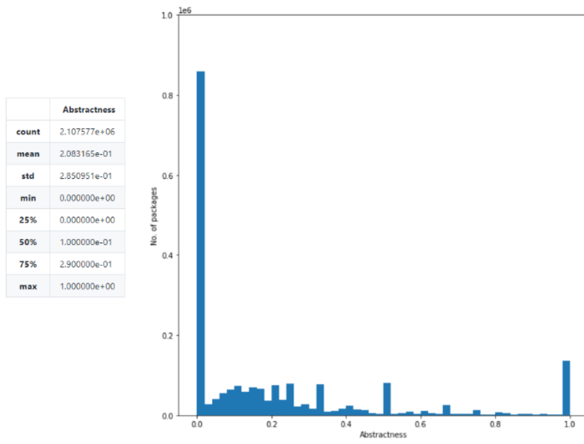
The distribution plots for these metrics in figure 3.25 to figure 3.28 show that 75% of the packages had up to 4 afferent couplings, most packages had $A = 0$, $I = 1$, DMM = 0. I sampled outliers based on the Ca metric. The 1,000 outlier packages had more than 191 afferent couplings with the maximum number of afferent couplings being 1,318. The 1,000 outlier packages contribute 267,379 afferent couplings out of the 8,796,206 afferent couplings contributed by all packages. Thus 0.0474% packages contribute 3.0397% of the total afferent couplings.

The sampled outliers were mostly core packages from projects like the AWS SDK for Java, Apache Zeppelin, Apache Hadoop Client API and JUnit. A common observation in the outliers for these metrics is that most packages were the core packages in their project. These core packages had many general classes that were used by the specialized classes in the specialized packages of their project. The number of afferent couplings of these packages was several times greater than the number of efferent couplings. As an example, package 'com.amazonaws' in project 'aws-java-sdk-bundle-1.11.99' is used by 360 packages including 'com.amazonaws.services.ec2', 'com.amazonaws.services.dynamodbv2', 'com.amazonaws.services.cloudwatch' etc. Each of these package dependencies holds classes for the various AWS services that require core classes from the 'com.amazonaws' package. The sampled outliers had different ratios of abstract and concrete classes.



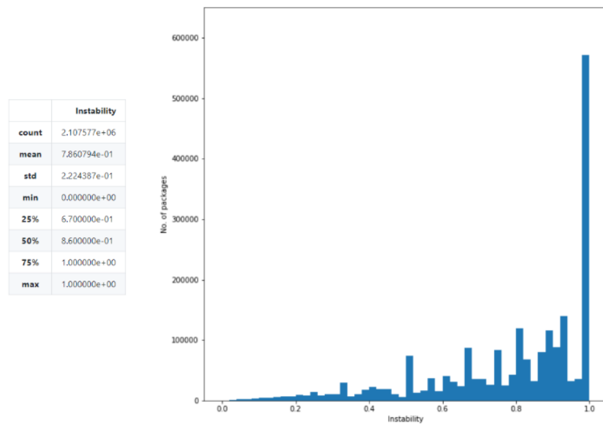
	Afferent Couplings
count	2.107577e+06
mean	4.173611e+00
std	1.126952e+01
min	0.000000e+00
25%	0.000000e+00
50%	1.000000e+00
75%	4.000000e+00
max	1.318000e+03

Figure 3.25 – Afferent Couplings



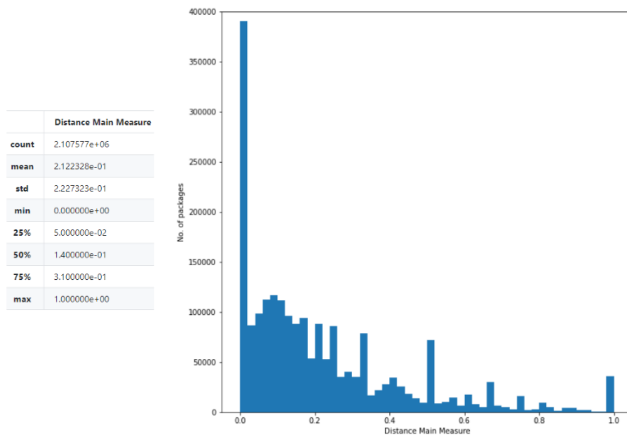
	Abstractness
count	2.107577e+06
mean	2.083165e-01
std	2.850951e-01
min	0.000000e+00
25%	0.000000e+00
50%	1.000000e-01
75%	2.900000e-01
max	1.000000e+00

Figure 3.26 – Abstractness



	Instability
count	2.107577e+06
mean	7.860794e-01
std	2.224387e-01
min	0.000000e+00
25%	6.700000e-01
50%	8.600000e-01
75%	1.000000e+00
max	1.000000e+00

Figure 3.27 – Instability



	Distance Main Measure
count	2.107577e+06
mean	2.122328e-01
std	2.227323e-01
min	0.000000e+00
25%	5.000000e-02
50%	1.400000e-01
75%	3.100000e-01
max	1.000000e+00

Figure 3.28 – Distance Main Measure

3.5 Overall observations from package level analysis

Like what was found in the class level analysis, a small number of outlier packages contribute a significant portion of the total number of classes and total number of afferent and efferent couplings. The inclusion or exclusion of these outliers in the training sets of machine learning models could have a significant impact on their performance. I also found correlations in some of these metrics which need to be considered too. Most outliers for the NC, NCC, NAC, Ce, Ca metrics were widely used packages linked with several other packages from their projects. Because of this large linkage, these should certainly be included in the training data sets for machine learning models. The metrics Ca, Ce and I for every package are related by a formula. The metrics A, I and DMM are also related in that DMM is the distance of a package from the line $A + I = 1$. These relationships must be taken into consideration to remove any redundant information.

Chapter 4

Conclusion

4.1 Limitations

The JAR files that I downloaded from the Maven Central Repository were based on the GroupID, ArtifactID and Version coordinates extracted from the Maven Central Index. I used the complete Maven Central Index generated in March 2020. Since then, more JAR files have been added to the Maven Central Repository which I did not consider. This limitation can be overcome by using the weekly increments of the Maven Central Index published since then. These weekly increments of the Maven Central Index include the coordinates of the newly added JARs and can be used to update the downloaded JAR file repository.

For every unique combination of GroupID and ArtifactID, I only downloaded the JAR file with the latest Version coordinate. This was done to speed up the download process and do an analysis based on the latest versions of different projects. However, this limits me from tracking the evolution of projects from version to version. This limitation can be overcome by trying to download multiple JARs in a single execution of the Maven Invoker.

There were some GroupID, ArtifactID and Version coordinates in the Maven Central Index for which there were no JAR files present in the Maven Central Repository. I have noted all such coordinates in a text file. For some JAR files, I could not execute CKJM extended and collect design metrics since the dependencies required to analyze these JAR files were not present in the Java classpath. I tried adding all the JAR files in the downloaded JAR file repository to the Java classpath to resolve this issue. However, some of the JAR files in the repository have an INDEX.LIST file which points to non-existent dependencies. This prevents me from adding all of them to the Java classpath. This limitation can be overcome by verifying the INDEX.LIST file corresponding to every JAR file in the repository.

4.2 Future Work

The scope of this work can be greatly expanded by including all the previous versions of every project in the Maven Central Repository. This would enable us to study the evolution of projects and study the changes in their design quality metrics over time. These changes in the design quality metrics can then be correlated with the code changes that led to a change in metric values. This would allow us to study design quality metric evolution in much greater detail.

Another area of future work could be to further refine the relationship between design quality metrics and software reusability, flexibility, understandability, functionality, extendibility and effectiveness based on the large empirical data obtained in this work.

4.3 Conclusion

I generated a repository containing the latest version of 226,793 software projects taken from the Maven Central Repository. I statically analyzed each of these projects and measured 18 class-level design quality metrics for 10,608,920 classes and 8 package-level design quality metrics for 2,107,577 packages. I analyzed the outlier projects in terms of object-oriented design quality metrics and evaluated their suitability for inclusion in training sets for machine learning models.

Bibliography

[1] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering,* vol. 28, no. 1, pp. 4-17, Jan. 2002.

[2] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target," *IEEE Software,* vol. 13, no. 1, pp. 12-21, Jan. 1996.

[3] J. A. McCall, P. K. Richards and G. F. Walters, Factors in Software Quality, Springfield, VA: National Tech. Information Service, 1977.

[4] G. R. Dormey, "A Model for Software Product Quality," *IEEE Trans. Softw. Eng.,* vol. 21, no. 2, p. 146–162, February 1995.

[5] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects.," in *5th International Conference on Dependability of Computer Systems*, 2010.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering,* vol. 20, no. 6, pp. 476-493, June 1994.

[7] M. Bunge, Treatise on Basic Philosophy: Ontology I: The Furniture of the World, Boston: Riedel, 1977.

[8] M. Bunge, Treatise on Basic Philosophy: Ontology II: The World of Systems, Boston: Riedel, 1979.

[9] N. Wilde and R. Huitt, "Maintenance support for object oriented programs," in *Conference on Software Maintenance*, Sorrento, Italy, 1991.

[10] G. Booch, Object oriented design with applications, Redwood City, CA, United States: Benjamin-Cummings Publishing, 1990.

[11] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering,* vol. 14, no. 9, pp. 1357-1365, 1988.

[12] S. Raemaekers, A. v. Deursen and J. Visser, "The Maven repository dataset of metrics, changes, and dependencies," in *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[13] "The Distributed ASCI Supercomputer 3," [Online]. Available: https://www.cs.vu.nl/das3/. [Accessed 28 October 2020].

[14] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios and D. Spinellis, "The Bug Catalog of the Maven Ecosystem," in *Association for Computing Machinery*, Hyderabad, India, 2014.

[15] V. Karakoidas, D. Mitropoulos, P. Louridas, G. Gousios and D. Spinellis, "Generating the Blueprints of the Java Ecosystem," *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories,* pp. 510-513, 2015.

[16] "JDepend - GitHub," [Online]. Available: https://github.com/clarkware/jdepend. [Accessed 28 October 2020].

[17] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas and I. Stamelos, "Evaluating the Quality of Open Source Software," in *International Workshop on Software Quality and Maintainability (SQM 2008)*, 2009.

[18] Wikipedia, "Apache Maven," [Online]. Available: https://en.wikipedia.org/wiki/Apache_Maven. [Accessed 28 October 2020].

[19] "Maven Central Repository," [Online]. Available: https://mvnrepository.com/repos/central. [Accessed 28 October 2020].

[20] "Maven Central Index," [Online]. Available: https://maven.apache.org/repository/central-index.html. [Accessed 28 October 2020].

[21] "Maven Indexer Core - Index Fields Reference," [Online]. Available: https://maven.apache.org/maven-indexer-archives/maven-indexer-LATEST/indexer-core/. [Accessed 28 October 2020].

[22] "Maven Indexer CLI," [Online]. Available: https://maven.apache.org/maven-indexer-archives/maven-indexer-LATEST/indexer-cli/. [Accessed 28 October 2020].

[23] "Luke - GitHub," [Online]. Available: https://github.com/DmitryKey/luke. [Accessed 28 October 2020].

[24] "ComparableVersion (Maven Artifact 3.0.3 API)," [Online]. Available: https://maven.apache.org/components/ref/3.0.3/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html. [Accessed 28 October 2020].

[25] "Apache Maven Dependency Plugin," [Online]. Available: http://maven.apache.org/plugins/maven-dependency-plugin/. [Accessed 28 October 2020].

[26] "Apache Maven Dependency Plugin - dependency:copy," [Online]. Available: http://maven.apache.org/plugins/maven-dependency-plugin/copy-mojo.html. [Accessed 28 October 2020].

[27] "Apache Maven Dependency Plugin - Usage," [Online]. Available: http://maven.apache.org/plugins/maven-dependency-plugin/usage.html. [Accessed 28 October 2020].

[28] "Apache Maven Invoker - Usage," [Online]. Available: https://maven.apache.org/shared/maven-invoker/usage.html. [Accessed 28 October 2020].

[29] "CKJM extended," [Online]. Available: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/. [Accessed 28 October 2020].

[30] "CKJM extended - GitHub," [Online]. Available: https://github.com/mjureczko/CKJM-extended. [Accessed 28 October 2020].

[31] "Apache Commons BCEL," [Online]. Available: https://commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/ClassFormatException.html. [Accessed 3 November 2020].

[32] Oracle, "JAR File Specification," [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Overview. [Accessed 3 November 2020].

[33] Wikipedia, "Software package metrics," [Online]. Available: https://en.wikipedia.org/wiki/Software_package_metrics. [Accessed 28 October 2020].

[34] "xmltodict - GitHub," [Online]. Available: https://github.com/martinblech/xmltodict. [Accessed 28 October 2020].

[35] "ONNX (Open Neural Network eXchange)," [Online]. Available: https://github.com/EmergentOrder/onnx-scala. [Accessed 16 November 2020].

[36] "Apache Derby," [Online]. Available: https://db.apache.org/derby/index.html. [Accessed 16 November 2020].

[37] "JOGL," [Online]. Available: https://jogamp.org/jogl/www/. [Accessed 16 November 2020].

[38] "Scalaz," [Online]. Available: https://github.com/scalaz/scalaz. [Accessed 16 November 2020].

[39] "AWS ObjectMapper," [Online]. Available: https://github.com/Netflix/awsobjectmapper. [Accessed 16 November 2020].

[40] "ZIO," [Online]. Available: https://github.com/zio/zio. [Accessed 16 November 2020].

[41] "Bouncy Castle," [Online]. Available: http://www.bouncycastle.org/java.html. [Accessed 16 November 2020].

[42] "JAXB v2," [Online]. Available: https://github.com/javaee/jaxb-v2. [Accessed 16 November 2020].

[43] "HAPI," [Online]. Available: https://hapifhir.github.io/hapi-hl7v2/index.html. [Accessed 16 November 2020].

[44] "Dekorate," [Online]. Available: https://github.com/dekorateio/dekorate. [Accessed 16 November 2020].

[45] "Kubernetes & OpenShift Java Client," [Online]. Available: https://github.com/fabric8io/kubernetes-client. [Accessed 16 November 2020].

[46] "SimpleFlatMapper," [Online]. Available: https://github.com/arnaudroger/SimpleFlatMapper. [Accessed 16 November 2020].

[47] "Frege," [Online]. Available: https://github.com/Frege/frege. [Accessed 16 November 2020].

[48] "Scala-js-dom," [Online]. Available: https://github.com/scala-js/scala-js-dom. [Accessed 16 November 2020].

[49] "Reactor Core," [Online]. Available: https://github.com/reactor/reactor-core. [Accessed 16 November 2020].

[50] "Eclipse SWT," [Online]. Available: https://www.eclipse.org/swt/. [Accessed 16 November 2020].

[51] "OJDBC 6," [Online]. Available: https://www.oracle.com/database/technologies/appdev/jdbc.html. [Accessed 16 November 2020].

[52] "Apache Sling," [Online]. Available: https://sling.apache.org/. [Accessed 16 November 2020].

[53] "Scala ISO," [Online]. Available: https://github.com/vitorsvieira/scala-iso. [Accessed 16 November 2020].

[54] "Apache Cassandra," [Online]. Available: https://cassandra.apache.org/. [Accessed 16 November 2020].

[55] "Google Ads API Client Library for Java," [Online]. Available: https://github.com/googleads/google-ads-java. [Accessed 16 November 2020].

[56] "Azure Management Libraries for Java," [Online]. Available: https://github.com/Azure/azure-libraries-for-java. [Accessed 16 November 2020].

[57] "AWS SDK for Java," [Online]. Available: https://github.com/aws/aws-sdk-java. [Accessed 16 November 2020].

[58] "Eclipse GlassFish," [Online]. Available: https://github.com/eclipse-ee4j/glassfish. [Accessed 16 November 2020].

[59] "Spring Framework," [Online]. Available: https://github.com/spring-projects/spring-framework. [Accessed 16 November 2020].

[60] "Apache Tapestry," [Online]. Available: https://github.com/apache/tapestry-5. [Accessed 16 November 2020].

[61] "JUnit 4," [Online]. Available: https://github.com/junit-team/junit4. [Accessed 16 November 2020].

Appendix

In this section I present the detailed metrics for a few of the sampled outliers for every metric.

1. Class Level Metrics

1.1 Weighted Methods per Class

Class 'org.emergentorder.onnx.package$QLinearConv' in project 'onnx-scala_2.13-0.2.0' has 13,139 methods. ONNX Scala is an open neural network exchange API, code generator and backend for functional deep learning in Scala [35]. This class has the highest number of methods all of which are public (WMC = NPM = 13,139) and most of them are static, synthetic or both. This also results in a high RFC and LOC (RFC = 13,148, LOC = 1,076,219). The average method complexity is very high (AMC = 80.9013). The class is right below the root and does not have any child classes (DIT = 1, NOC = 0). This results in zero functional abstraction (MFA = 0). It has a very high amount of coupling (CBO = 17, Ca = 0, Ce = 17) but also a high degree of cohesion in its methods (LCOM = 86,310,091, LCOM3 = 2.0, CAM = 0.8563). LCOM suggests that no single pair of methods formed from the 13,139 methods accesses a common attribute. LCOM3 suggests that some attributes of this class are not accessed by some methods. However, the high value of CAM suggests that most methods work with the same method parameter types resulting in cohesion. Since this class does not inherit methods from any class, there is no coupling between inherited and class methods (IC = CBM = 0). None of the class attributes are private or user defined class types (DAM = 0.0, MOA = 0).

Class 'org.apache.derby.impl.sql.compile.SQLParser' in project 'derby-all-10.8.1.2' has 993 methods. Apache Derby is an open source relational database implemented in Java [36]. This class has 385 public methods and in response to any message, an object of this class could potentially invoke many methods (WMC = 993, NPM = 385, RFC = 1181). This also results in many lines of code (LOC = 61,784). The average method complexity is on the higher side (AMC = 61.1339). The class is just below the root and does not have any

children (DIT = 1, NOC = 0). This results in zero functional abstraction (MFA = 0). It has a high amount of coupling (CBO = 75, Ca = 1, Ce = 74) and a severe lack of cohesion in its methods (LCOM = 385,626, LCOM3 = 0.9155, CAM = 0.0331). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). Many attributes are private and 12 are user defined class types (DAM = 0.8325, MOA = 12).

1.2 Number of Public Methods

Class 'javax.media.opengl.TraceGL' in project 'jogl-1.1.1' has 2,097 public methods. JOGL hosts the Java Binding for the OpenGL API and provides hardware supported 3D graphics to applications written in Java [37]. This class is made up of 2,101 methods of which 2,097 are public. Any object of this class could potentially call 4,216 methods (WMC = 2,101, NPM = 2,097, RFC = 4,216). Every method either returns void or some Java primitive type. Every method parameter is either a Java primitive or belongs to 'java.nio'. As a result, this class does not have to depend on any other class (CBO = 0, Ca = 0, Ce = 0) and the methods are highly cohesive (LCOM = 0, LCOM3 = 0.0013, CAM = 0.1005). However, the average method complexity is much higher (AMC = 33.3660). This is also reflected in the number of lines of code (LOC = 72,206). This class is right below the root of the object hierarchy and does not have any children (DIT = 1, NOC = 0). Since no methods other than those of the root are inherited, functional abstraction is zero (MFA = 0.0). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). All attributes of this class are private and are user defined types (DAM = 1.0, MOA = 1).

1.3 Response for a Class

Class 'scalaz.Scalaz' in project 'scalaz-core_2.11.0-M7-7.1.0-M4' can call up to 950 methods in response to a message. Scalaz is a Scala library for functional programming [38]. This class has 475 methods all of which are public but note that the number of methods that could be called is double the number of methods in this class (WMC = NPM = 475, RFC = 950). This must mean that this class uses many methods from other classes. This is shown

by the coupling metrics for this class which indicate that it depends upon 168 other classes but is not used by any of them (CBO = 168, Ca = 0, Ce = 168). There is also a severe lack of cohesion among the methods (LCOM = 112,575, LCOM3 = 2.0, CAM = 0.0139). This class is right below the root of the object hierarchy and does not have any child classes (DIT = 1, NOC = 0). It has no functional abstraction (MFA = 0). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). It has a typical average method complexity (AMC = 4.5263). Due to the large number of methods the number of lines of code is large (LOC = 2,625). The class has no private or user defined class type attributes (DAM = 0.0, MOA = 0).

1.4 Average Method Complexity

Class 'com.netflix.awsobjectmapper.AmazonObjectMapper' in project 'awsobjectmapper-1.9.33.1' has an average method complexity of 7,889.0. AWS Object Mapper can be used to convert AWS model object to/from JSON [39]. This class has one public method which is a constructor, and an object of this class could call up to eight methods in response to a message (WMC = NPM = 1, RFC = 8). The single method has a significantly higher method complexity than typical (AMC = 7,889.0). This large method complexity can be explained by the several lines of code (LOC = 7,890). This class is four levels below the root of the object hierarchy and has no child classes (DIT = 4, NOC = 0). All methods other than the constructor have been inherited from the parents which results in full functional abstraction (MFA = 1.0). This class is coupled with six other classes (CBO = 6, Ca = 0, Ce = 6) and since it does not have any attributes of its own, the cohesion metrics indicate that its methods are highly cohesive (LCOM = 0, LCOM3 = 2.0, CAM = 1.0). Since it does not have any methods of its own, there is no coupling with any parent classes through inheritance (IC = 0, CBM = 0). None of its attributes are private or user defined class types (DAM = 0.0, MOA = 0).

1.5 Lack of Cohesion in Methods

Class ‘zio.Chunk’ in project ‘zio_2.12-1.0.0-RC9-4’ has an LCOM of 10,344,426. ZIO is a Scala library for asynchronous and concurrent programming [40]. This class has 4,549 methods of which 4,547 are public and any object of this class could call up to 4,646 methods in response to a message (WMC = 4,549, NPM = 4,547, RFC = 4,646). The average method complexity is typical (AMC = 7.4929). This class is right below the root of the object hierarchy and has no child classes (DIT = 1, NOC = 0). This class calls methods from many other classes and is also called by the methods of many other classes (CBO = 85, Ca = 55, Ce = 44). There is a severe lack of cohesion among its methods (LCOM = 10,344,426, LCOM3 = 2.0, CAM = 0.1388). This class has one method which is coupled with every method inherited from the parent class (IC = 1, CBM = 1) (the parent class is the root of the object hierarchy in this case). Since the only methods inherited are those defined in the root, it has zero functional abstraction (MFA = 0.0). It also has several lines of code (LOC = 38,634) and does not have any private or user defined class type attributes (DAM = 0.0, MOA = 0).

1.6 Lack of Cohesion in Methods 3

Class ‘org.bouncycastle.asn1.smime.SMIMECapabilityVector’ in package ‘bcprov-jdk14-1.65’ has LCOM3 = 0.0. Bouncy Castle is a lightweight cryptography API for Java [41]. This class has five methods all of which are public. An object of this class could potentially invoke 12 methods (WMC = NPM = 5, RFC = 12). All methods have a high method complexity (AMC = 12.2). The class is right below the root of the object hierarchy and has no child classes (DIT = 1, NOC = 0). It is coupled with six other classes which is typical (CBO = 6, Ca = 1, Ce = 5). However, the methods of this class are highly cohesive. Three methods work with ‘org.bouncycastle.asn1.ASN1ObjectIdentifier’ objects, and one method returns a ‘org.bouncycastle.asn1.ASN1EncodableVector’ (LCOM = 0, LCOM3 = 0.0, CAM = 0.5). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). Since no methods other than those of the root are inherited, functional abstraction is zero (MFA = 0.0). This class has 67 lines of code which is on the higher side but is typical (LOC = 67). All attributes of this class are private and are user defined class types (DAM = 1.0, MOA = 1).

1.7 Cohesion Among Methods of Class

Class 'com.sun.msv.reader.relax.core.RELAXCoreReader$StateFactory' in project 'jaxb1-impl-2.2.5.1' has a CAM score of 0.9524. Java Architecture for XML Binding provides an API and tools that automate the mapping between XML documents and Java objects [42]. This class has 14 methods of which 13 are protected and one is public. An object of this class could call up to 31 methods in response to a message (WMC = 14, NPM = 1, RFC = 31). Average method complexity is 4.9286 which is typical (AMC = 4.9286). This class is two levels below the root of the object hierarchy and has one child class (DIT = 2, NOC = 1). It has high functional abstraction (MFA = 0.35). It is highly coupled (CBO = 22, Ca = 6, Ce = 17) however the values for cohesion are conflicting (LCOM = 89, LCOM3 = 0.9231, CAM = 0.9524). LCOM3 suggests that the methods of this class lack cohesion, but CAM suggests otherwise. All methods of this class take 'com.sun.msv.reader.State' and 'com.sun.msv.util.StartTagInfo' as parameters. CAM considers method parameter types, whereas LCOM3 considers class attributes accessed by methods. This may explain the conflicting results. None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). All attributes of this class are private, and one is a user defined class type (DAM = 1.0, MOA = 1).

1.8 Coupling Between Object Classes

Interface 'ca.uhn.hl7v2.parser.ModelClassFactory' in project 'hapi-osgi-base-2.3' is coupled with 6,149 other classes. HL7 is a messaging specification for healthcare information systems. HAPI provides an open source HL7 parser for Java [43]. This interface also has one of the highest values of afferent coupling (CBO = 6149, Ca = 6148, Ce = 2). This large value of coupling can be explained by the fact that this class is a method parameter in many other classes in the same project. As it is an interface, all methods are abstract and public (WMC = NPM = 6, RFC = 6). As a result, the average method complexity is zero (AMC = 0.0). This interface is right below the root of the object hierarchy and is not extended by any other

interface (DIT = 1, NOC = 0). As a result, the functional abstraction is zero (MFA = 0). This interface does not have any attributes as a result of which LCOM and LCOM3 suggest that there is no cohesion. However, CAM suggests that there is a small amount of cohesion (LCOM = 15, LCOM3 = 2.0, CAM = 0.6250). The interface has only six lines of code, one for each of the six methods (LOC = 6). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). There are no static attributes in this interface (DAM = 0, MOA = 0).

1.9 Afferent Couplings

Interface 'io.dekorate.deps.kubernetes.api.builder.Nested' in project 'dekorate-dependencies-0.9.9' is used by 5,180 other classes. Dekorate is a collection of Java compile-time generators and decorators for Kubernetes manifests [44]. This interface has a single public and abstract method (WMC = NPM = 1, RFC = 1). Since it is an interface, method complexity is zero (AMC = 0.0). This interface is right below the root of the object hierarchy and is not extended by any other interface (DIT = 1, NOC = 0). Since no methods other than those of the root are inherited, functional abstraction is zero (MFA = 0.0). This interface has been implemented by many other classes in its project and is thus highly coupled (CBO = 5,180, Ca = 5,180, Ce = 0). Since there is only one method and no attributes, the cohesion metrics indicate that it is highly cohesive (LCOM = 0, LCOM3 = 2.0, CAM = 1.0). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). The only method in this interface occupies a single line of code (LOC = 1). There are no static attributes in this interface (DAM = 0.0, MOA = 0).

1.10 Efferent Couplings

Interface 'io.fabric8.kubernetes.api.model.WatchEventFluent' in project 'crd-generator-0.17.0' uses 1,702 other classes. 'fabric8io/Kubernetes-client' is a Java client for Kubernetes and OpenShift REST APIs [45]. This interface comprises of 1,702 public and abstract methods many of which return objects implementing WatchEventFluent or many

of its nested interfaces (WMC = NPM = 1,702, RFC = 1,702). Since it is an interface average method complexity is zero (AMC = 0.0). It is right below the root of the object hierarchy and has no children (DIT = 1, NOC = 0). Since no methods other than those of the root are inherited, functional abstraction is zero (MFA = 0.0). WatchEventFluent has several nested interfaces and nearly every method of this interface either returns an object that implements WatchEventFluent or one of its nested interfaces. This results in a very high amount of coupling (CBO = 1,591, Ca = 534, Ce = 1,058). There is a severe lack of cohesion in the methods of this interface (LCOM = 1,447,551, LCOM3 = 2.0, CAM = 0.0032). None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). Every method in this interface occupies a single line of code (LOC = 1,702). There are no static attributes in this interface (DAM = 0.0, MOA = 0).

1.11 Depth of Inheritance Tree

Class 'org.simpleflatmapper.tuple.Tuple32' in project 'sfm-tuples-8.2.1' is 31 levels below the root of the object hierarchy and has no children (DIT = 31, NOC = 0). SimpleFlatMapper is a lightweight alternative to Hibernate and provides fast parsing and mapping of CSV files to Java POJOs [46]. This class has 5 methods all of which are public, and an object of this class could potentially call many methods in response to a message (WMC = NPM = 5, RFC = 47). The methods have a higher complexity (AMC = 53.0). The higher depth in the inheritance tree also results in a very high functional abstraction (MFA = 0.9758). The class has low coupling (CBO = 1, Ca = 1, Ce = 1) and its methods are very cohesive (LCOM = 0, LCOM3 = 0.0, CAM = 0.7). However, this class has 13 methods which are coupled with every inherited method, and it is coupled with 9 parent classes through inheritance, which is much higher than typical (IC = 9, CBM = 13). The class is rather large with many lines of code (LOC = 271). The class attributes are all private and none of them are user defined class types (DAM = 1.0, MOA = 0).

1.12 Number of Children

Class ‘frege.runtime.Fun1’ in project ‘frege-3.22.367-g2737683’ has 9,101 children. Frege is a functional programming language that compiles to Java and runs on the JVM [47]. This class is two levels below the object root and has the highest number of children classes (DIT = 2, NOC = 9,101). It has 5 methods all of which are public, and any object of this class could call up to 6 methods in response to a message which is typical (WMC = NPM = 5, RFC = 6). The methods have a typical method complexity (AMC = 3.0). This class has a rather high functional abstraction (MFA = 0.5556). It also has a very high amount of coupling (CBO = 10,181, Ca = 10,178, Ce = 4) and less cohesion among its methods (LCOM = 10, LCOM3 = 2, CAM = 0.6). However, none of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). The class has 20 lines of code (LOC = 20) and has no private or user defined class type attributes (DAM = 0, MOA = 0).

1.13 Measure of Functional Abstraction

Class ‘org.scalajs.dom.raw.PerformanceMeasure’ in project ‘scalajs-dom_sjs1.0.0-M1_2.13.0-M1-0.9.3’ has a functional abstraction of 1.0. ‘scala-js-dom’ is a statically typed DOM API for Scala.js [48]. This class has a single public constructor, and any object of this class can call up to 2 methods in response to a message (WMC = NPM = 1, RFC = 2). The average method complexity is typical (AMC = 3.0). This class is three levels below the root of the object hierarchy and has no child classes (DIT = 3, NOC = 0). This class has a functional abstraction of 1.0 which implies that all methods other than the constructor are inherited from parent classes (MFA = 1.0). It is coupled with only one other class which is typical (CBO = 1, Ca = 0, Ce = 1). Since it does not have any methods of its own, there is no coupling with any parent classes through inheritance (IC = 0, CBM = 0). Since this class does not have any attributes, the cohesion metrics suggest that the methods of this class are highly cohesive (LCOM = 0, LCOM3 = 2.0, CAM = 1.0). The class has only four lines of code (LOC = 4) and does not have any attributes which are private or user defined class types (DAM = 0.0, MOA = 0).

1.14 Inheritance Coupling

Class 'reactor.util.function.Tuple8' in project 'reactor-core-3.3.4.RELEASE' is coupled with 7 parent classes through inheritance. 'reactor-core' is a non-blocking Reactive Streams Foundation for the JVM [49]. This class has 42 methods, 41 of which are public. An object of this class could call up to 22 methods in response to a message (WMC = 42, NPM = 41, RFC = 22). The average method complexity is slightly higher than typical (AMC = 9.833). This class is seven levels below the root of the object hierarchy and has no child classes (DIT = 7, NOC = 0). It has a high functional abstraction (MFA = 0.7405). It is coupled with eight other classes (CBO = 8, Ca = 2, Ce = 6) and there is a lack of cohesion in its methods (LCOM = 679, LCOM3 = 0.5244, CAM = 0.4762). This class has 23 methods which are coupled with every inherited method from the seven parent classes which is much higher than typical (IC = 7, CBM = 23). It has 457 lines of code (LOC = 457) and half of its attributes are private, but none of them are user defined class types (DAM = 0.5, MOA = 0).

1.15 Coupling Between Methods

Class 'org.eclipse.swt.widgets.Shell' in project 'org.eclipse.swt.gtk.linux.x86_64-4.3' has 157 methods coupled with all inherited methods. Eclipse SWT is an open source widget toolkit for Java [50]. This class has 116 methods of which 45 are public. An object of this class could call up to 388 methods in response to a message. (WMC = 116, NPM = 45, RFC = 388). The average method complexity is much higher than typical (AMC = 49.1466). This class is seven levels below the root of the object hierarchy and has no children (DIT = 7, NOC = 0). It has very high functional abstraction (MFA = 0.8519). This class has a very high coupling (CBO = 105, Ca = 74, Ce = 39) and a severe lack of cohesion among its methods (LCOM = 3,630, LCOM3 = 0.9486, CAM = 0.1121). It has 157 methods which are coupled with every inherited method, and it is coupled with 6 parent classes through inheritance, which is much higher than typical (IC = 6, CBM = 157). This class has 5,840 lines of code (LOC = 5,840) and none of its attributes are private, but two of them are user defined class types (DAM = 0.0, MOA = 2).

1.16 Lines of Code

Class 'oracle.jdbc.driver.PhysicalConnection' in project 'ojdbc6_g-11.2.0.4' has 65,430 lines of code. OJDBC6 is a database driver compatible with JDK 6, 7 and 8 [51]. This class has 408 methods of which 312 are public and any object of this class could potentially call up to 744 methods in response to a message (WMC = 408, NPM = 312, RFC = 744). The average method complexity is very high (AMC = 158.8260). This class is three levels below the root of the object hierarchy and has three child classes (DIT = 3, NOC = 3). It has a high functional abstraction (MFA = 0.3447). It also has a very high amount of coupling with other classes (CBO = 179, Ca = 94, Ce = 99) and a severe lack of cohesion among methods (LCOM = 0, LCOM3 = 0.9846, CAM = 0.0259). LCOM = 0 can be possibly explained by there being an equal number of method pairs which access a common attribute and method pairs which do not access a common attribute. This class has 13 methods which are coupled with every inherited method from the three parent classes which is much higher than typical (IC = 3, CBM = 13). Very few of the class attributes are private and 18 attributes are user defined class types (DAM = 0.0362, MOA = 18).

1.17 Data Access Metric

Class 'org.apache.sling.junit.impl.servlet.XmlRenderer' in project 'org.apache.sling.junit.core-1.0.8' has a ratio of 0.8182. Apache Sling is a framework for RESTful web-applications based on an extensible content tree [52]. This class has 20 methods of which 19 are public and an object of this class could potentially call up to 84 methods in response to a message (WMC = 20, NPM = 19, RFC = 84). The average method complexity is very high (AMC = 21.65). This class is two levels below the root of the object hierarchy and has no child classes (DIT = 2, NOC = 0). It has a high functional abstraction (MFA = 0.3214). It is coupled with seven other classes which is typical (CBO = 7, Ca = 0, Ce = 7) but there is a lack of cohesion in its methods (LCOM = 144, LCOM3 = 0.9091, CAM = 0.1545). None of the methods are coupled with any parent class through inheritance (IC =

0, CBM = 0). This class is very large with 464 lines of code (LOC = 464). Most of the class attributes are private but none of them are user defined class types (DAM = 0.8182, MOA = 0).

1.18 Measure of Aggregation

Scala singleton object class 'com.vitorsvieira.iso.ISOCountrySubdivision' in project 'scala-iso_2.11-0.1.2' extends multiple Scala traits like 'ISOCountrySubdivisionAfrica', 'ISOCountrySubdivisionAsia', 'ISOCountrySubdivisionEurope' etc. These Scala traits contain multiple fields of type 'ISOCountrySubdivision' for every country subdivision in the world. 'scala-iso' stores ISO related Scala types for country codes, country subdivisions, country currency etc. [53]. In Scala, a singleton object is a class with only one instance. Traits are used to share interfaces and fields between classes and are like interfaces in Java. The class 'ISOCountrySubdivision' has 6,857 methods of which 6,856 are public and any object of this class could potentially call up to 6,883 methods (WMC = 6,857, NPM = 6,856, RFC = 6,883). These methods are getters and setters for every country subdivision in the world. The average method complexity is high (AMC = 14.0047). This class is right below the root of the object hierarchy and does not have any child classes (DIT = 1, NOC = 0). The class is coupled with many other classes (CBO = 34, Ca = 4, Ce = 33) and there is a severe lack of cohesion in its methods (LCOM = 22,640,196, LCOM3 = 0.9996, CAM = 0.3001). This lack of cohesion can be explained by the fact that there is a getter and setter for every country subdivision. None of the methods are coupled with any parent class through inheritance (IC = 0, CBM = 0). Most attributes are of type 'com.vitorsvieira.iso.CountrySubdivision' where every attribute of such type represents one of the many country subdivisions of the world. All attributes of such type are private and have public getters and setters. This results in very high values for DAM and MOA (DAM = 0.9997, MOA = 3,424). The large number of attributes and their getter and setter methods also result in an increase in the number of lines of code (LOC = 106,366).

2. Package Level Metrics

2.1 Number of Classes

Package ‘org.apache.cassandra.thrift’ in project ‘cassandra-thrift-3.9’ has 1,004 classes. Apache Cassandra is a highly scalable distributed database [54]. This package has 1,004 classes of which 1,001 are concrete and 3 are abstract. This package is not used by any other package in the project, but uses 13 other packages (Ca = 0, Ce = 13). It uses packages like ‘java.io’ and ‘org.apache.thrift’. It has 0 abstractness and 1 instability resulting in a DMM of 0 (A = 0, I = 1, DMM = 0).

Package ‘com.google.ads.googleads.v2.resources’ in project ‘google-ads-5.0.0’ has 1,196 concrete classes. ‘google-ads-java’ is the Google Ads API client library for Java [55]. This package has 1,196 classes of which 1,039 are concrete and 157 are abstract. This package is used by 3 other packages in the project, and uses 13 other packages (Ca = 3, Ce = 13). It is used by ‘com.google.ads.googleads.v2.services’, ‘com.google.ads.googleads.v2.utils’ etc. and uses packages like ‘com.google.common.base’, ‘com.google.ads.googleads.v2.common’ etc. It has 0.13 abstractness and 0.81 instability resulting in a DMM of 0.06 (A = 0.13, I = 0.81, DMM = 0.06).

Package ‘com.microsoft.azure.management.containerregistry.v2018_02_01_preview’ in project ‘azure-mgmt-containerregistry-1.0.0-beta-1’ has 101 abstract classes. ‘azure-libraries-for-java’ is a set of Azure management libraries for Java [56]. This package has 163 classes of which 62 are concrete and 101 are abstract. This package is used by 1 other package in the project, and uses 12 other packages (Ca = 1, Ce = 12). It is used by ‘com.microsoft.azure.management.containerregistry.v2018_02_01_preview.implementation’, and uses packages like ‘com.microsoft.azure.arm.collection’, ‘com.microsoft.azure.arm.model’ etc. It has 0.62 abstractness and 0.92 instability resulting in a DMM of 0.54 (A = 0.62, I = 0.92, DMM = 0.54).

2.2 Afferent Couplings

Package ‘com.amazonaws’ in project ‘aws-java-sdk-bundle-1.11.99’ is used by 360 other packages. ‘aws-sdk-java’ is the official AWS SDK for Java [57]. This package has 33 classes of which 25 are concrete and 8 are abstract. This package is used by 360 other packages in the project, and uses 22 other packages (Ca = 360, Ce = 22). It is used by ‘com.amazonaws.services.ec2’, ‘com.amazonaws.services.dynamodbv2’, ‘com.amazonaws.services.cloudwatch’ and many other packages, and uses packages like ‘com.amazonaws.auth’, ‘com.amazonaws.regions’ etc. It has 0.24 abstractness and 0.06 instability resulting in a DMM of 0.7 (A = 0.24, I = 0.06, DMM = 0.7).

2.3 Efferent Couplings

Package ‘com.sun.enterprise.web’ in project ‘glassfish-embedded-all-5.1.0’ uses 130 other packages. Eclipse GlassFish is a Jakarta EE compatible implementation by the Eclipse Foundation [58]. This package has 67 classes of which 53 are concrete and 14 are abstract. This package is used by 19 other packages in the project, and uses 130 other packages (Ca = 19, Ce = 130). It is used by ‘com.sun.enterprise.security.webservices’, ‘org.glassfish.webservices’ etc. and uses packages like ‘org.glassfish.internal.api’, ‘javax.servlet.http’, ‘com.sun.enterprise.web.connector’ and many others. It has 0.21 abstractness and 0.87 instability resulting in a DMM of 0.08 (A = 0.21, I = 0.87, DMM = 0.08).

2.4 Abstractness

Package ‘org.springframework.web.portlet.bind.annotation’ in project ‘spring-webmvc-portlet-4.3.9.RELEASE’ has an abstractness of 1. Spring Framework makes it easier to create enterprise Java applications [59]. This package has 4 classes, all of which are abstract. This package is used by 1 other package in the project, and uses 4 other packages

(Ca = 1, Ce = 4). It is used by ‘org.springframework.web.portlet.mvc.annotation’ and uses packages like ‘org.springframework.core.annotation’ etc. ‘ActionMapping’, ‘EventMapping’, ‘RenderMapping’ and ‘ResourceMapping’ are the four abstract classes. It has 1 abstractness and 0.8 instability resulting in a DMM of 0.8 (A = 1, I = 0.8, DMM = 0.8).

2.5 Instability

Package ‘org.apache.tapestry5.jmx’ in project ‘tapestry-jmx-5.5.0’ has an instability of 0.5. Apache Tapestry is a component-oriented framework for web applications in Java [60]. This package has one abstract class. This package is used by 2 other packages in the project, and uses 2 other packages (Ca = 2, Ce = 2). It is used by ‘org.apache.tapestry5.internal.jmx’ and ‘org.apache.tapestry5.jmx.modules’ and uses packages ‘java.lang’ and ‘javax.management’. It has 1 abstractness and 0.5 instability resulting in a DMM of 0.5 (A = 1, I = 0.5, DMM = 0.5).

2.6 Distance Main Measure

Package ‘org.junit.internal.requests’ in project ‘junit-4.9’ has a DMM of 0.0. JUnit is a unit testing framework for Java [61]. This package has 4 classes of which 3 are concrete and 1 is abstract. This package is used by 2 other packages in the project, and uses 6 other packages (Ca = 2, Ce = 6). It is used by ‘org.junit.runner’ and ‘org.junit.experimental.max’ and uses packages like ‘org.junit.internal.builders’, ‘org.junit.internal.runners’ etc. It has 0.25 abstractness and 0.75 instability resulting in a DMM of 0.0 (A = 0.25, I = 0.75, DMM = 0.0).