Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Identifying refactoring opportunities for large packages by analyzing maintainability characteristics in Java OSS[☆]

Haris Mumtaz[a,*], Paramvir Singh[b], Kelly Blincoe[a]

[a] Department of Electrical, Computer, and Software Engineering at the University of Auckland, New Zealand
[b] School of Computer Science at the University of Auckland, New Zealand

ARTICLE INFO

Article history:
Received 4 June 2022
Received in revised form 12 March 2023
Accepted 18 April 2023
Available online 29 April 2023

Keywords:
Package maintainability
Package smells
Package metrics
Empirical analysis
Open-source software

ABSTRACT

The source code of a Java-based software system is often structured into packages. When packages are large, they often carry maintainability quality issues. In the literature, there is a lack of empirical evidence on the specific maintainability issues that occur when packages become too large. Our study fills this gap by performing relationship analysis of package size with respect to internal maintainability characteristics (coupling, cohesion, and complexity) using package-level metrics collected from 111 open-source Java projects provided in *Qualitas Corpus*. Our results show significantly higher maintainability issues in large packages as indicated by the maintainability metrics. We also report strong relationships of package size with cohesion (represented by the *number of connected components* in a package) and complexity (measured by the *number of internal relationships* in a package). Based on these strong associations with package size, we show that these cohesion and complexity metrics can be used to identify large package refactoring opportunities. Furthermore, we also discuss why some maintainability metrics (e.g., coupling metrics) may not be useful for refactoring large packages.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Software developers on large software systems need to work in parallel and often independently. Splitting the software into modules (subsystems) is one way to enable this independent development (Parnas, 1972). Building a software system based on a non-modular design can cause issues when the size of the system becomes large (Parnas, 1972). Therefore, a high-level abstraction design is required that can structure a system into modules. In Java software development, the source code is structured into classes, which are further grouped into "packages" (Almugrin et al., 2016). A package may be created to group semantically related classes or to represent a component (module or subsystem) of a system (Lippert and Roock, 2006). The purpose of collecting the classes into packages is to have a clean structure of the classes for easy access and better maintainability (Lippert and Roock, 2006). Therefore, the packages should be structured in such a way that the package size is maintainable (Lippert and Roock, 2006).

Software smells are sub-optimal structural patterns in the software and are often indicators of technical debt (Falessi and Kazman, 2021). If smells are not removed, debt will continue to accumulate causing high maintenance costs (Malakuti and Heuschkel, 2021). Software smells can occur at different levels of granularity—architecture, design, and code. Package smells reflect the sub-optimal patterns at the architecture level (i.e., package structure) of a software product (Lippert and Roock, 2006). One of the package smells is the "large package", which relates to the size characteristic. The number of classes (regardless of their visibility and abstractness) in a package is a common measure for package size (Lippert and Roock, 2006). In case a package has too many classes, where maintaining the package becomes a problem, the package has a large package smell (Lippert and Roock, 2006). The problem of the large package is similar to the "large class" code smell (Beck et al., 1999). A large class is likely to have many associations with other classes because of the large number of method calls (Beck et al., 1999); similarly, a large package could have many associations (i.e., coupling) with other packages. On a similar note, classes that are not communicating with each other within a package show that such classes have different responsibilities (Ampatzoglou et al., 2019). These issues are related to the insufficient modularization smell because abstractions (in this case, packages) are not fully decomposed (Suryanarayana et al., 2015). Due to the lack of proper decomposition, the size of the packages may not be manageable and can also cause

☆ Editor: Christoph Treude.
* Corresponding author.
E-mail addresses: hmum126@aucklanduni.ac.nz (H. Mumtaz), p.singh@auckland.ac.nz (P. Singh), k.blincoe@auckland.ac.nz (K. Blincoe).

https://doi.org/10.1016/j.jss.2023.111717
0164-1212/© 2023 Elsevier Inc. All rights reserved.

comprehension issues. According to Suryanarayana et al. (2015), the effective modularization is to "decompose abstractions (packages) to manageable size". This suggests that package size should be maintained to avoid issues like insufficient modularization in the package structure. However, the literature lacks in providing the empirical evidence for identifying refactoring opportunities for package size optimization.

Previous package-level research has investigated techniques to improve the internal quality characteristics, such as coupling and cohesion (Abdeen et al., 2009; Ampatzoglou et al., 2019; Almugrin et al., 2016; Chantian and Muenchaisri, 2019). For instance, techniques have been proposed to improve package structure by managing the package coupling and cohesion (Abdeen et al., 2009; Ampatzoglou et al., 2019), predicting package maintainability and testability using coupling metrics (Almugrin et al., 2016), and refactoring large packages using coupling-based impacts (unstable dependencies) (Chantian and Muenchaisri, 2019). Most of these techniques focus on improving the coupling between the packages and enhancing cohesion within the packages through "move class" refactoring. However, having only good coupling or good cohesion in the package structure does not mean that package size is maintainable. In addition, there are other issues, such comprehension and communication issues, that are associated with large packages (Parnas, 1972; Lippert and Roock, 2006). Therefore, identifying refactoring opportunities focused on package size is equally important.

Larger entities in the software (e.g., large package, large class, large method) are often associated with more maintainability problems (Parnas, 1972). In the context of this study, the size measure (i.e., number of classes) only reveal the large packages in the software and not sufficient to explain how the refactoring of large packages should be approached. Therefore, additional measures (metrics) are needed to assist the refactoring process. For example, connected components (*ConnComp*), a package cohesion metric, indicates the number of disjoint clusters of connected classes in a package. If a large package has more connected components, its size can be reduced by splitting the package into multiple packages, with each component in a separate package (Lippert and Roock, 2006; Palomba et al., 2015). In this example, the connected components (*ConnComp*) metric assisted the refactoring of large packages, which size metric alone cannot guide. However, not all maintainability metrics could be used for identifying refactoring opportunities. Therefore, how do we know what are the maintainability metrics that could be useful for refactoring large packages? This can be achieved by examining the relationships between maintainability metrics and package size metric because those metrics that are strongly associated with package size can be considered for package size improvement.

To the best of our knowledge, there is a lack of empirical evidence on identifying refactoring opportunities focused on package size and investigating the package metrics for such purpose. Therefore, in this study, we investigate the relationships between package size and internal maintainability characteristics (using previously-employed and well-known coupling, cohesion, and complexity metrics) with the objective of identifying the strongly associated metrics with package size. The benefit of such analysis is that the maintainability metrics that are strongly related with package size can be used as indicators of maintainability issues in large packages because a single threshold for large package identification may not be accurate in every case. We performed a preliminary analysis to show that, in our dataset, packages with almost similar size (i.e., having slightly different number of classes) have different maintainability metrics, meaning exhibiting different maintainability issues. We select 30 classes in a package as a base because the literature assumes that when the number of classes in a package exceeds 30, the package can be considered large (Martin, 2002). Therefore, the other packages for the analysis will be either with 29 classes or 31 classes. After selecting all the packages with these number of classes (29, 30, and 31), we computed the mean values of the maintainability metrics. The mean values of the maintainability metrics are listed in Table 1. A coupling metric (*Ce*) indicates that on average the coupling issues were less when the packages have 31 classes as compared to packages with 29 classes. Furthermore, the cohesion (as indicated by *ConnComp* metric) got worse as the package size increased from 29 classes to 31 classes. Finally, first, the average complexity (indicated by *R* metric) increased, when classes increased (i.e., from 29 to 30) in the packages; whereas, the average complexity got better when another class was added to the packages (i.e., from 30 to 31). This preliminary analysis on our dataset suggests that packages with almost similar sizes can behave differently by exhibiting different maintainability issues. Therefore, only relying on package size for refactoring may not be always beneficial and measurements beyond size will be useful for impactful refactoring. Moreover, the metrics can help identifying appropriate refactoring opportunities for package size decomposition. Before we perform such analysis, using the metrics, we empirically show the extent of the maintainability issues (associated with different package sizes) to express why refactoring of large packages is needed in the first place. Although it is commonly understood and theoretically discussed that large components in the software are usually more prone to maintainability issues, there is no empirical evidence that shows the extent of the maintainability issues associated with large packages. Therefore, before identifying refactoring of large packages, we empirically show whether and to what extent large packages suffer from maintainability issues.

Table 1
Varied behavior of maintainability metrics in marginally different-sized large packages.

Package size	Coupling[a]		Cohesion[a]		Complexity[a]
	Ce	*Ca*	*H*	*ConnComp*	*R*
29 classes (21 packages)	25.76	53.09	1.52	11.19	49
30 classes (21 packages)	25.47	21.23	2.23	11.28	71.2
31 classes (22 packages)	21.36	22.18	1.67	17.45	56

[a]The definition of the metrics can be found in Table 3.

Our empirical investigation is driven by two research questions:

RQ1 — *What is the extent of maintainability issues (as indicated by the maintainability metrics) with respect to different package sizes (large, moderate, and small)?*

RQ2 — *What are the maintainability characteristics (metrics) useful for identifying refactoring opportunities for large packages?*

To answer RQ1, we examine the differences across large, moderate, and small packages in the internal maintainability characteristics measured using all of the package-level metrics proposed by Martin (2002) (measuring coupling, cohesion, and complexity—also listed in Table 3). The differences are examined using the *Kruskal–Wallis* test, descriptive statistics (central tendency statistics—min, max, mean, and median), and risk ratio. This analysis explains the severity of the maintainability issues in relation to the concrete package size thresholds (i.e., large, moderate, and small packages). The classification of large, moderate, and small packages is identified using distribution analysis. The threshold-based size classification is only used in RQ1 because it is needed to show the extent of the maintainability issues associated with different package sizes, primarily with large packages. To address RQ2, we investigate the linear relationship of package size with maintainability characteristics by performing correlation analysis, regression analysis, and effect size measurement using the same package-level metrics (presented by Martin (2002)). The empirical analysis is conducted using a collection of 111 open-source Java projects compiled in *Qualitas Corpus* (Terra et al., 2013).

Based on our results, for RQ1, we observe that the maintainability issues (indicated by the metrics) are significantly higher in large packages in comparison with moderate and small packages. This indicates that large packages carry significantly more maintainability issues. In terms of RQ2, although the regression results indicate a strong association of package size with the three maintainability characteristics (coupling, cohesion, and complexity metrics), we observe that package cohesion (represented by the number of connected components formed by the classes or interfaces in the package—*ConnComp* metric Martin, 2002) and package complexity (measured through the number of relationships between classes/interfaces in the package—*R* metric Martin, 2002) have the greatest effect. Therefore, these maintainability metrics can be more useful for identifying the refactoring opportunities for large packages. It is also worth mentioning that, theoretically, coupling is considered to be degraded in the large entities of a software (large packages, large class, long method); however, our results reveal the contrary (i.e., large packages do not suffer from additional coupling).

The main contributions of our empirical analysis are as follows:

1. Our analysis contributes to the refactoring aspect of the package structure by identifying the usefulness of package cohesion (connected components—*ConnComp*) and complexity (number of relationships—*R*) metrics for refactoring large packages using a well-known set of Java projects compiled in *Qualitas Corpus*.
2. A replication package[1] containing package-level metrics (in *CSV* format) of 111 open-source Java projects from the *Qualitas Corpus* dataset is provided. The package also includes the python script used for our data analysis. Our replication package can also be useful in future research to analyze the quality (size, coupling, cohesion, and complexity) of Java packages.

2. Related work

Previous research has substantially studied the maintainability of software systems, covering various artifacts (e.g., architecture, design, and source code). The maintainability of software design and source code has been intensively researched, describing several methods for examining the relationships between different internal maintainability characteristics (size, coupling, cohesion, and complexity). Since the analysis methods (for design and source code) can be applicable for software architecture, we first describe the previous work related to the maintainability of software design and source code (Section 2.1). Next, we describe the literature on the maintainability of object-oriented architecture by presenting the techniques focusing on software packages (Section 2.2).

2.1. Design and code maintainability

Several researchers have investigated the maintainability in software design and source code using internal quality characteristics, such as size, coupling, cohesion, complexity (Al Dallal, 2013; Briand et al., 1993; Li and Henry, 1993; Aggarwal et al., 2008; Zhou and Leung, 2007; Elish and Elish, 2009; Dagpinar and Jahnke, 2003; Rizvi and Khan, 2010). Al Dallal (2013) empirically studied the relationship between internal class quality characteristics (size, cohesion, and coupling) and class maintainability using statistical methods, such as descriptive statistics, univariate regression, and multivariate regression. Li and Henry (1993) also investigated class-level quality metrics for inheritance, coupling, cohesion, complexity, and size to build prediction models for maintainability. The results of these studies suggested a strong correlation between the quality metrics and class maintainability. Likewise, using the data from Li and Henry's (1993) prior study, other studies employed different statistical methods to develop maintainability prediction models (Aggarwal et al., 2008; Zhou and Leung, 2007; Elish and Elish, 2009). These studies also reached the same conclusion as by Li and Henry (1993).

Similarly, Dagpinar and Jahnke (2003) applied univariate and multivariate regression analysis to predict class maintainability using object-oriented metrics related to size, inheritance, cohesion, and coupling. Their results showed that size and efferent coupling measures are significantly correlated with maintainability, suggesting that these measures can be used for refactoring large classes. Furthermore, Rizvi and Khan (2010) explored the relationship between maintainability and internal characteristics (e.g., size and complexity) and reported a high correlation between them. Lastly, there is some literature that discussed various metrics suitable for analyzing the maintainability quality (related to coupling, cohesion, and complexity) at the design-level of software projects (Lanza and Marinescu, 2007; Bansiya and Davis, 2002).

It can be observed that various statistical analysis, especially regression analysis, have been applied to investigate the relationship of different maintainability aspects to identify refactoring opportunities at the design and code level. In this study, we also examine such relationships to identify refactoring opportunities for large packages using multiple statistical methods (including regression analysis) by analyzing Java packages.

2.2. Architecture maintainability

Some techniques have been proposed to analyze the package maintainability by investigating the package-level quality aspects, such as coupling and cohesion. For instance, Abdeen et al. (2009) proposed a technique to improve package coupling and cycles by optimizing the dependencies between packages. Their technique indirectly modified the package sizes (within the predefined constraints) by moving the classes among packages to have better coupling (using efferent coupling (*Ce*) and afferent coupling (*Ca*) of packages) and cohesion (dependencies between classes of a package). Another study also used efferent coupling (*Ce*) and afferent coupling (*Ca*) to modularize the package structure using coupling (dependencies between classes of the packages) (Ampatzoglou et al., 2019). Chantian and Muenchaisri (2019) refactored large packages using community detection. Their technique used efferent (*Ce*) and afferent (*Ca* couplings only to monitor the impact of refactoring on the coupling aspect. Almugrin et al. (2016) predicted package maintainability and testability using coupling-related metrics. e Abreu and Goulao (2001) applied a clustering-based technique to populate software modules (packages) with classes having high cohesion, and non-relation classes were extracted from the packages. Furthermore, Briand et al. (1993) reported a strong association between package-level coupling metrics (efferent (*Ce*) and afferent (*Ca*)) with project maintainability. Finally, a couple of methods have been proposed that are based on genetic algorithms to distribute classes into packages so that the internal package dependencies (cohesion) can be enhanced (Seng et al., 2005; Harman et al., 2002). It can be seen that, in terms of package coupling analysis, the proposed techniques have used the metrics (efferent (*Ce*) and

[1] https://figshare.com/s/714099c8686d05e88631.

afferent (*Ca* couplings) to modularize the software packages (Abdeen et al., 2009; Ampatzoglou et al., 2019). The techniques reduced the efferent (*Ce*) and afferent (*Ca*) couplings by moving classes between the packages (i.e., reducing dependencies between packages). While these techniques focused on modularizing packages, they did not specifically aim to reduce package size; therefore, large packages are still possible.

Although the improvement of the coupling characteristic indirectly changes the package size, the coupling metrics have not been explicitly studied for identifying their relationship with package size. In addition, it may be possible that both the coupling metrics (efferent—*Ce* and afferent—*Ca*) show different levels of association with package size. Therefore, the strengths of the relationships of package size with both the coupling metrics individually need to be established. In this study, we investigate the relationship of package size with coupling characteristic measured using efferent coupling (*Ce*) and afferent coupling (*Ca*) metrics.

Similarly, cohesion improvement methods also indirectly impact the package size; however, the existing techniques have not studied their relationship. Similar to the coupling aspect, Chantian and Muenchaisri (2019) applied the concept of communities measured using cohesion metrics—relational cohesion (*H*) and connected components (*ConnComp*)—however, they only assessed the impact of refactoring large packages on the cohesion aspect. Based on the cohesion metrics, they employed Louvain algorithm (Blondel et al., 2008) to identify communities (each community represented by a group of classes having strong cohesion) in the relationship graph of a package. Briand et al. (1993) studied the relationship of package-level cohesion metrics with project maintainability and reported their strong association. Their cohesion metrics are similar to relational cohesion (*H*) and connected components (*ConnComp*); however, they used different terminologies (e.g., ratio of cohesive interactions). In this study, we examine the relationship of cohesion characteristic with package size using cohesion metrics (*ConnComp* and *H*).

Lastly, only one of the existing package improvement techniques considered the complexity aspect (Seng et al., 2005). Seng et al. (2005) focused on subsystems decomposition by optimizing metrics and heuristics based on a good modularized design. In addition, they adopted a different metric suitable for computing the complexity of subsystems. In our study, we analyze the relationship of complexity characteristic (measured using *R* metric) with package size. Lastly, besides only relying on structural aspects to remodularize the software packages, Mkaouer et al. (2015) used various factors, such as structural, semantical, and history-based, to remodularize the package structure. Their multi-objective search-based method help improving the package structure while minimizing the changes and maintaining the semantic information.

Overall, it can be observed that most of the package maintainability techniques focus on enhancing the package coupling and cohesion aspects. However, identifying refactoring opportunities focused on package size and investigating the package metrics for such purpose are not empirically studied. Our analysis fills this gap through a relationship study of package size with three maintainability aspects (coupling, cohesion, and complexity) by examining package-level metrics collected from a large set of open-source Java projects.

3. Empirical study design

Our study design is driven by the experimental guidelines provided by Wohlin et al. (2012) and Runeson and Höst (2009). In this section, we first describe our study scope followed by our study plan.

Table 2
Statistics of the projects under analysis.

Projects	111
Versions	752
Packages	7156
Classes	Min – 0 Mean – 9.82 Max – 1091 Total – 70,331
Methods	Min – 0 Mean – 98.2 Max – 23,684 Total – 7,02,849
Domains	Tool – 30 Middleware – 19 Testing – 11 Visualization – 10 Parser – 9 Database – 8 Graphics – 7 SDK – 6 IDE – 5 Games – 3 Programming – 3
Status	Active – 76.5% Dormant – 20.7% Inactive – 2.7%

3.1. Study scope

The goal of our empirical study is to *analyze the maintainability metrics* for the purpose of *identifying refactoring opportunities for large packages* with respect to *package maintainability* from the viewpoint of *software development* in the context of *open-source software projects.*

3.2. Study plan

In this section, we describe the objects (i.e., open-source projects), variables (i.e., metrics), variables collection (i.e., metrics collection), and data analysis methods of our empirical study.

3.2.1. Open-source projects

In this study, we use the *Qualitas Corpus* (Terra et al., 2013) (originally curated by Tempero et al. (2010)) that provides a collection of 111 compiled open-source Java projects (having 752 versions in total). The corpus can also be found on GitHub.[2] *Qualitas Corpus* has 16,509 packages, over 200,000 compiled classes, and above 18.5 million lines of code. The projects in the corpus are of varying sizes and belong to different domains (e.g., tool, middleware, parser, IDE, etc.). In addition, the status of majority of the projects is *active*—there are some projects that are in *Dormant* state (i.e., temporarily inactive). The statistics of the dataset used in this study are presented in Table 2. The corpus has been used in many studies to analyze software quality and the relationship among metrics (Sales et al., 2013; Sousa et al., 2017), which is also accomplished in this study (i.e., investigating the relationships between package size and maintainability characteristics (metrics)); therefore, making the *Qualitas Corpus* suitable for our analysis. The smallest project has one package, and the largest one contains 1508 packages (over 5100 classes and 40*k* methods). For some projects, the *Qualitas Corpus* has multiple versions; in such cases, we select the first (base) version of the project in our analysis.

[2] https://github.com/JavaQualitasCorpus.

Table 3
Description of the package-level metrics considered in this study.

Metric	Attribute	Description
Number of classes (*NumCls*)	Size	Number of classes in the package (Lippert and Roock, 2006).
Number of operations in the classes (*NumOpsCls*)[a]	Size	Number of operations (methods) in the classes of the package (Lorenz and Kidd, 1994).
Number of classes in sub-packages (*NumCls_tc*)[a]	Size	Number of classes in the packages and its sub-packages (Lippert and Roock, 2006).
Efferent coupling (*Ce*)	Coupling	Number of external classes/interfaces that the package depends on Martin (2002).
Afferent coupling (*Ca*)	Coupling	Number of external classes/interfaces that depend on the package (Martin, 2002).
Relational cohesion (*H*)	Cohesion	Average number of internal relationships per class/interface in the package (Martin, 2002).
Connected components (*ConnComp*)	Cohesion	Number of connected components formed by the classes/interfaces of the package (Martin, 2002).
Number of relationships (*R*)	Complexity	Number of relationships between classes/interfaces that are internal to the package (Martin, 2002).

[a]Metric used as a control variable in the regression model.

3.2.2. Metrics

To investigate the relationships between package size and internal maintainability characteristics (coupling, cohesion, and complexity), package-level variables are required that measure these aspects. Therefore, we employ all the package-level metrics (related to coupling, cohesion, and complexity) presented by Martin (2002). To represent the package size, we employ the metric by Lippert and Roock (2006). For control variables, we use package size metrics presented by Lorenz and Kidd (1994). There are other popular maintainability metrics presented by Lanza and Marinescu (2007) and Bansiya and Davis (2002); however, they measure the design-level quality aspects (e.g., quality at the class-level). Therefore, these metrics are not fitted for our analysis. The package-level metrics we select are also suitable because they measure the quality characteristics we are interested in and have been used successfully in the previous studies (Abdeen et al., 2009; Ampatzoglou et al., 2019; Almugrin et al., 2016; Chantian and Muenchaisri, 2019). Both classes and interfaces, where applicable, are considered for computing the package metrics. The package size is represented by the number of classes (*NumCls*) in the package—the more the classes in the package, the bigger the package will be. Coupling is measured using two standard measures, efferent coupling (*Ce*) and afferent coupling (*Ca*). Complexity is measured using the number of relationships (*R*) metric that is based on the concept of Control (Information) Flow Metric between software entities (e.g., modules, classes, methods) (Henry and Kafura, 1981). Cohesion is considered using the relational cohesion (*H*) and connected components (*ConnComp*) metrics. Here "component" represents a graph of connected classes and interfaces in a package. Relational Cohesion (*H*) considers the number of relationships (*R*) metric to compute the internal relationships in the package. We include two additional metrics (independent variables) related to package size—number of operations in the classes (*NumOpsCls*) and number of classes in sub-packages (*NumCls_tc*)—as control variables in our regression model. The package metrics are briefly explained in Table 3.

3.2.3. Metrics collection

To collect the required metrics, we employ two software tools. We use *Enterprise Architect*,[3] a popular tool that can model, design, and test software systems (Matthes et al., 2008; Anacleto, 2008), to reverse-engineer the source code of software projects to obtain a high-level representation of packages and classes. Next, we use *SDMetrics*,[4] a widely employed tool to measure software design characteristics (Samli et al., 2020; Briand and Wüst, 2002; Briand et al., 1998). The rationale of choosing *SDMetrics* is the support provided to collect the metrics of our interest. The following steps (also depicted in Fig. 1) are involved in the collection of the required metrics:

1. Reverse-engineer the source code of a project using *Enterprise Architect* to obtain a high-level project representation because we are interested in package structure.
2. Export the high-level representation of the project's source code in XMI format (required format for *SDMetrics*) using the *Enterprise Architect's* export functionality.
3. Upload the XMI of the project to the *SDMetrics* tool.
4. Calculate and export (in *CSV* format) the package-level metrics of the project.

3.2.4. Data analysis methods for RQ1

To examine the maintainability metrics (related to coupling, cohesion, and complexity) with respect to different package sizes, we first classify large, moderate, and small packages. The literature assumes that if the number of classes in a package exceeds 30, the package is large (Martin, 2002). Another study reported that more than 28 classes in a package is a large package (Filó et al., 2015). In addition, a package with only a few classes (either one or two) is considered a small package (Martin, 2002). However, to be confident that our package size classifications hold true for the projects analyzed in this study, we classify the large, moderate, and small packages by applying distribution analysis on the number of classes (*NumCls*) in the packages of 111 open-source Java projects.

Many techniques analyzed the distribution of the metric values of the benchmark data (set of software projects) to filter the values that are uncommonly used by the developers (Alves et al., 2010; Aniche et al., 2016; Ferreira et al., 2012; Filó et al., 2015; Fontana et al., 2015; Le et al., 2018; Mori et al., 2018; Vale et al., 2019). Such uncommon values were used as a break-point to classify low-quality components in the software. For instance, Alves et al. (2010) used quantiles to identify the metrics values for different quality levels (high, moderate, and low). Although the objective of our study is not to report any thresholds, we adopt the quantile-based technique (by Alves et al. (2010)) only for classifying package sizes (large, moderate, and small) for this study. We applied the quantile-based technique by looking at the distribution of the number of classes in the packages (*NumCls* metric) across the 111 open-source Java projects in our dataset. Alves et al. (2010) reported 90% quantile as very high risk; therefore, we also use the 90% quantile as a break-point to filter the packages with high risk (i.e., large packages). Low quantile values capture the small metric values with very low variability (Alves et al., 2010); therefore, we use the 25% quantile to classify the packages with only a few classes (i.e., small packages). Everything in between (25%–90%) are considered moderate packages.

Statistical Test and Descriptive Statistics. Using the internal maintainability metrics, we perform *Kruskal–Wallis H*-tests and compute descriptive statistics. Each *H*-test has a trio sample (e.g., cohesion values from large, moderate, and small packages) to identify whether the difference is significant or not. Moreover, the descriptive statistics provide some indication of the differences in the maintainability issues associated with large,

[3] https://sparxsystems.com/products/ea/index.html.
[4] https://www.sdmetrics.com/index.html.

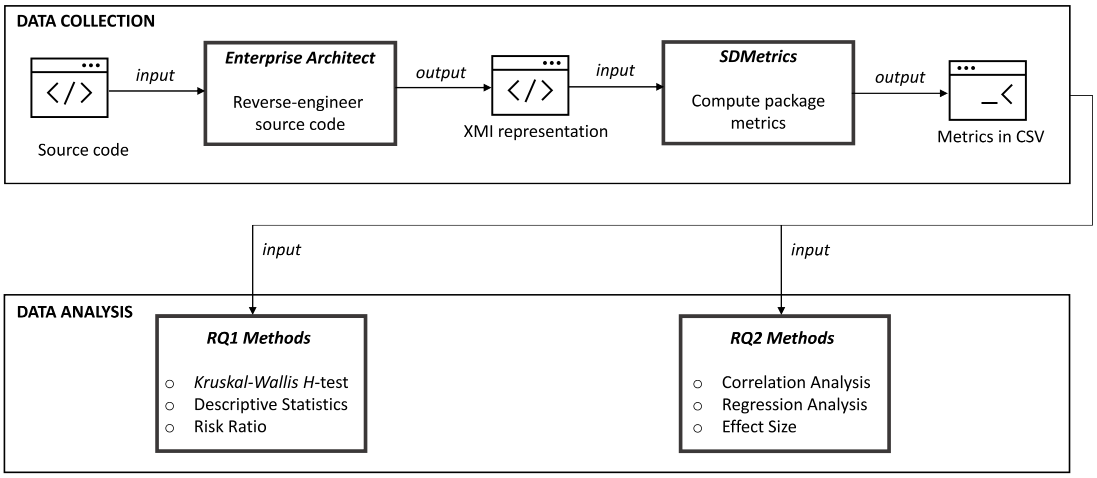

Fig. 1. Study design.

moderate, and small packages. For all the maintainability metrics (except relational cohesion (H)), high values represent bad quality. Therefore, if large packages have high values of the maintainability metrics, it will suggest that such packages have more maintainability issues.

Risk Ratio. In the case of coupling metrics, we perform additional analysis based on the threshold values cataloged by Filó et al. (2015). The coupling thresholds, given by Filó et al. (2015), are also based on the *Qualitas Corpus*—the set of projects used in this study. The thresholds are proposed in terms of good, bad, and regular coupling ranges. The "good" coupling range corresponds to the most commonly used coupling values in practice; "bad" range encapsulates the coupling values that are uncommonly adopted by the developers; and, finally, "regular" coupling range represents the values that are neither too frequent nor rare (Filó et al., 2015). The thresholds are listed in Table 4. Based on these threshold ranges, we calculate the risk ratios (RR) for both of the coupling measures (efferent (Ce) and afferent (Ca))—in terms of good, bad, and regular—using Eq. (1) (presented by Zhang and Kai (1998)).

$$RR = \frac{CI_{l(bad)}}{CI_{s(bad)}} \quad (1)$$

where $CI_{l(bad)}$ is the cumulative incidence of the large packages having coupling in bad range, whereas, $CI_{s(bad)}$ represents the cumulative incidence of the small packages having coupling in bad range. $CI_{l(bad)}$ and $CI_{s(bad)}$ are calculated using Eqs. (2) and (3), respectively.

$$CI_{l(bad)} = \frac{Number\ of\ large\ packages\ with\ bad\ coupling}{Total\ number\ of\ large\ packages} \quad (2)$$

$$CI_{s(bad)} = \frac{Number\ of\ small\ packages\ with\ bad\ coupling}{Total\ number\ of\ small\ packages} \quad (3)$$

Using the same Eqs. (1), (2), and (3), the risk ratios are calculated for good and regular ranges of coupling metrics; and also, for moderate packages. Since we could not find the threshold ranges for cohesion and complexity metrics (employed in this study), we could not compute the risk ratio for these characteristics.

3.2.5. Data analysis methods for RQ2

Correlation Analysis. To identify the linear relationships between package size and three maintainability characteristics (coupling, cohesion, and complexity), we calculate the correlation using the package-level metrics. However, to decide which correlation method is suitable for analysis, we test whether our data is normally distributed or not. Therefore, we apply *Shapiro–Wilk test* for checking the normality of our dataset. The output of the *Shapiro–Wilk test* suggests that the distribution of our dataset is not normally distributed. The results of the test are summarized in Table 5. Since our data is not normally distributed, we compute the Spearman's correlation (Prion and Haerling, 2014). According to Prion and Haerling (2014), the Spearman's correlation (strength of the relationship) is interpreted as follows: 0–0.20 as *negligible*, 0.21–0.40 as *weak*, 0.41–0.60 as *moderate*, 0.61–0.80 as *strong*, and 0.81–1.00 as *very strong*.

Regression Analysis. We build a regression model to show the significance of the relationships between package size (dependent variable) and maintainability metrics (independent variables) related to coupling, cohesion, and complexity. We also include two control variables (as described in Section 3) in our regression model. Before performing the regression analysis, we check whether any collinearity exists between the independent variables. For that, we calculate the Variance Inflation Factor (VIF) of the independent variables (including control variables). VIF checks if the behavior of an independent variable is influenced by its correlation with the other independent variables in the regression model. The VIFs are computed in pairs (shown as a matrix in Table 6) and collectively (presented in Table 7). We find that the VIF values are small, suggesting that no such collinearity exists between the independent variables; therefore, we build the regression model.

Table 4
Thresholds for classifying different coupling quality levels.

Metric	Good	Bad	Regular
Efferent coupling (Ce)	≤ 6	>16	$6 < Ce \leq 16$
Afferent coupling (Ca)	≤ 7	>39	$7 < Ca \leq 39$

Table 5
Shapiro–Wilk test for normality distribution.

Metric	Coefficient
Number of classes (*NumCls*)	0.25***
Efferent coupling (*Ce*)	0.51***
Afferent coupling (*Ca*)	0.21***
Connected components (*ConnComp*)	0.32***
Relational cohesion (*H*)	0.61***
Number of relationships (*R*)	0.16***
Number of operations in classes (*NumOpsCls*)[a]	0.15***
Number of classes in sub-packages (*NumCls_tc*)[a]	0.13***

*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$
[a]Metric used as a control variable in the regression model.

Table 6
Pairwise Variance Inflation Factor (VIF) for identifying collinearlity between a pair of metrics.

Metric	*Ce*	*Ca*	*ConnComp*	*H*	*R*	*NumOpsCls*	*NumCls_tc*
Efferent coupling (*Ce*)	–	1.16	1.32	1.31	1.18	1.25	1.06
Afferent coupling (*Ca*)	–	–	1.07	1.15	1.17	1.12	1.07
Connected components (*ConnComp*)	–	–	–	1.05	1.10	1.69	1.07
Relational cohesion (*H*)	–	–	–	–	1.38	1.13	1.03
Number of relationships (*R*)	–	–	–	–	–	2.09	1.07
Number of operations in classes (*NumOpsCls*)[a]	–	–	–	–	–	–	1.08
Number of classes in sub-packages (*NumCls_tc*)[a]	–	–	–	–	–	–	–

[a]Metric used as a control variable in the regression model.

Table 7
Collective Variance Inflation Factor (VIF) scores for identifying collinearlity between all the metrics.

Metric	VIF
Efferent coupling (*Ce*)	1.71
Afferent coupling (*Ca*)	1.31
Connected components (*ConnComp*)	2.13
Relational cohesion (*H*)	1.66
Number of relationships (*R*)	2.93
Number of operations in classes (*NumOpsCls*)[a]	3.57
Number of classes in sub-packages (*NumCls_tc*)[a]	1.15

[a]Metric used as a control variable in the regression model.

Effect Size. To show the strength of the relationships of the maintainability metrics in our regression model, we express their global and local effect sizes using Cohen's f^2 measure (Selya et al., 2012). The importance of reporting effect size is that it is more practical and independent of the sample size (Sullivan and Feinn, 2012). The global effect size is computed using the following equation (where R^2 is the coefficient of determination):

$$f^2 = \frac{R^2}{1 - R^2} \tag{4}$$

Another variation of Cohen's f^2 measures the local effect size of each independent variable in the regression model. Since we have multiple maintainability metrics (variables), we also compute their local effect sizes using the following equation:

$$f^2 = \frac{R^2_{AB} - R^2_A}{1 - R^2_{AB}} \tag{5}$$

where B is the variable of interest, such as coupling metric (Ce), cohesion metric (H), etc., and A is the set of all other variables, depending on what B is. For interpreting the effect size, according to Cohen's guidelines (Cohen, 2013), $f^2 \geq 0.02$ represents *small* effect size; $f^2 \geq 0.15$ represents *medium* effect; and $f^2 \geq 0.35$ represents a *large* effect size.

4. Analysis results

In this section, we present the results of our empirical analysis in terms of our RQs.

4.1. Maintainability issues in different package sizes (RQ1)

The distribution of the number of classes in the packages is presented in Fig. 2 (outliers are excluded to ease readability). It can be seen that the distribution is long-tail, where most of the packages have a small number of classes. By specifying the 90% quantile, we receive the metric (number of classes in the package—*NumCls*) value 22. Therefore, we consider any package having classes more than 22 as a large package. The red line (on the right in Fig. 2) shows the break-point (i.e., 22 classes in the package at 90% quantile). The plot shows that the high-risk metric values (i.e., beyond 90%—right side of the red line) rarely occur

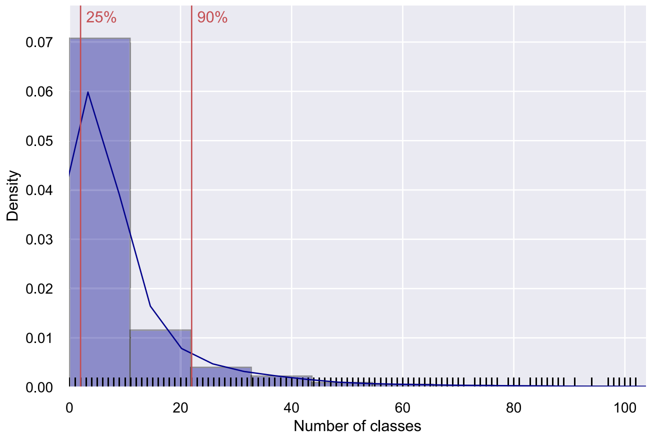

Fig. 2. Distribution of the number of classes in the packages of the analyzed projects. The vertical red lines are drawn at 25% quantile (number of classes is 2) and 90% quantile (number of classes equals 22). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in the projects. Similarly, at 25% quantile, the yielded value is 2, which means that 25% of the packages contain either one or two classes. Therefore, we consider any package having one or two classes as a small package. To show this in Fig. 2, another red line (left side) is drawn at 25% quantile. To summarize, by analyzing the distribution of package sizes, the large, moderate, and small packages are identified as follows:

- **Large package:** Number of classes in the package $>$ 22.
- **Small package:** Number of classes in the package $=$ 1 OR 2.
- **Moderate package:** Number of classes in the package $>$ 2 AND $\leq$ 22.

By applying these package classification rules, out of total of 7156 packages, we obtain 695 large (comprises 36,156 classes), 4048 moderate (contains 31,005 classes), and 2242 small (has 3170 classes) packages. The remaining 171 are packages with no classes. These 171 packages have only interfaces that is why they are counted as empty in terms of number of classes. We also validated this by manually looking at the quarter of these 171 packages to confirm that these 171 packages actually have interfaces only. It is worth mentioning that although 695 large packages (out of 7156) is a small percentage, the classes within these large packages are more than moderate and small packages combined, suggesting a large number of classes will be affected by the maintainability issues associated with large packages. The descriptive statistics of the maintainability metrics, in Table 8, indicate more maintainability issues (except in relational cohesion (H)) in large packages than in small packages because high values represent low maintainability. For instance, in large packages, afferent coupling (Ca) has maximum and median values significantly higher than that in moderate and small packages. In

Table 8
Descriptive statistics of maintainability metrics with respect to large, moderate, and small packages.

Metric	*Ideal* value	*H*-test	Large packages			Moderate packages			Small packages		
			Min	Max	Median	Min	Max	Median	Min	Max	Median
Efferent coupling (*Ce*)	Lower	1825.04***	0	192	18	0	240	5	0	73	1
Afferent coupling (*Ca*)	Lower	1185.35***	0	1488	15	0	760	1	0	609	0
Connected components (*ConnComp*)	Lower	2918.14***	1	393	13	1	56	3	1	55	1
Relational cohesion (*H*)	Higher	169.17***	0.01	23.1	1.6	0.03	20.5	1	0.18	7	1
Number of relationships (*R*)	Lower	2809.99***	0	4815	64	0	534	5	0	140	0

*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$.

the case of relational cohesion (H), the median value is slightly lower in moderate and small packages, meaning the relational cohesion (H) is better in large packages. We performed additional analysis of relational cohesion (H) in large and small packages to further understand this behavior. First, based on the number of classes (more than 22), we found that there are 695 large packages in our dataset. After that, we found those instances where the relational cohesion (H) is 1 or higher (suggesting that the package has good internal communication). As a results, out of 695 large packages, 458 packages exhibit good relational cohesion (H). This means that as the number of classes is increasing in the packages, the communication between the classes within the packages is getting better (i.e., developers are adding mostly those classes that are needed in the package). Similarly, we selected 928 small packages (having 2 classes) and found that 587 packages have no internal communications (i.e., their relational cohesion (H) is worse). This means that the classes in small packages are working independently. One possible reason for this could be that small packages are being created to store classes that do not quite "fit" into any package, this could be a reason for them not having any internal communication. Future work could perform qualitative analysis to understand the reasons for this more deeply. Our *Kruskal–Wallis* tests also show the significant differences in the maintainability metrics (for coupling, cohesion, and complexity) when comparing different package sizes. In all the cases, we obtain high H-scores and p-values less than 0.001 (see Table 8), indicating the statistically significant differences in the maintainability measures of large, moderate, and small packages.

The risk ratios of the coupling metrics (in good, bad, and regular ranges) are described as follows (also presented in Table 9):

- **Efferent Coupling (*Ce*):** For good efferent coupling, we receive risk ratios of 0.28 (for moderate packages) and 0.19 (in terms of small packages), suggesting a significantly low likelihood that large packages have good efferent coupling (see Table 9). On the other hand, the likelihood of bad efferent coupling in large packages is nearly five times (*RR* value is 4.78) higher compared to moderate packages and many-fold (*RR* value is very high—43.89) compared to small packages. Finally, similar likelihood behavior is observed in regular coupling (see Table 9).
- **Afferent Coupling (*Ca*):** Similar behavior is observed in the afferent coupling (*Ca*), where we obtain a risk ratio of 0.47 and 0.4 for moderate and small packages, respectively, meaning large packages have a lower likelihood of having good afferent coupling (*Ca*) (see Table 9). On the contrary, the risk ratio of bad afferent couplings (5.3 and 21.89 for moderate and small, respectively) suggests a significantly higher likelihood of bad afferent coupling in large packages. Likewise, the risk ratio for regular afferent coupling indicates over four times (*RR* is 4.52) more coupling risk in large packages than in small packages (see Table 9).

Large packages have significantly higher values of coupling, cohesion, and complexity metrics. This indicates maintainability issues in large packages and call the need for their refactoring.

4.2. Identifying metrics useful for refactoring large packages (RQ2)

Our regression model produces an R^2 value of 0.81 (with a p-value less than 0.001 and intercept 3.09—see Table 10, showing the overall significance of the relationship between package size and maintainability metrics (coupling, cohesion, and complexity). Based on the R^2, the global effect size (computed using Eq. (4)) of the regression model is 4.37. Since effect size (f^2) is greater than 0.35 (Cohen, 2013), the global effect size is significantly large. However, we notice that the individual relationships and effect sizes (calculated using Spearman's correlation and local effect size, respectively) of package size with these maintainability metrics show varied behavior (described in the rest of this section). The results are also summarized in Table 12.

4.2.1. Size and coupling

Spearman's correlations suggest moderate and weak relationships of efferent coupling (*Ce*) and afferent coupling (*Ca*), respectively, with package size. The correlations of coupling metrics are listed in Table 11. However, when controlled for the number of operations in the classes (*NumOpsCls*) and the number of classes in the sub-packages (*NumCls_tc*), our regression model shows negligible associations of coupling metrics with package size, given their small coefficient values and no effect sizes (see Table 10). It is also worth noting that the afferent coupling (*Ca*) (that has a weak correlation) has negative coefficient in the multiple regression model. As explained by Falk and Miller (1992), this behavior occurs when the relationship between the dependent and independent variables is weak that the difference in the signs reflects random variation around zero.

4.2.2. Size and cohesion

Spearman's correlations indicate that the cohesion (measured by the number of connected components—*ConnComp*) is strongly associated with package size, whereas, for the other cohesion (computed by the average number of internal relationships—H), the relationship is negligible. The correlations can be found in Table 11. Our regression analysis confirms the strong relationship of cohesion (*ConnComp*) with package size because of its high coefficient value and medium effect size (see Table 10). However, the coefficient of relational cohesion (H) in the regression analysis is negative, which reflects its linear decrease in relation to the package size increase. The negative coefficient for relational cohesion (H) practically makes more sense because internal package relationships (between classes and interfaces) are decreasing (i.e., low relational cohesion) as the package size increases (Martin, 2002). However, the effect size of relational cohesion (H) is small (f^2 is 0.03).

Table 9
Risk ratios of couplings in large packages with respect to moderate and small packages.

Coupling	Risk ratios w.r.t. moderate packages			Risk ratios w.r.t small packages		
	Good	Bad	Regular	Good	Bad	Regular
Efferent coupling (*Ce*)	0.28***	4.78***	1.01***	0.19***	43.89***	4.16***
Afferent coupling (*Ca*)	0.47***	5.30***	1.85***	0.4***	21.89***	4.52***

$^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$.

Table 10
Regression analysis and effect size measurements.

Metric	Coefficient	Effect size (f^2)
Efferent coupling (*Ce*)	0.045***	0.00
Afferent coupling (*Ca*)	−0.005*	0.00
Connected components (*ConnComp*)	0.802***	0.24
Relational cohesion (*H*)	−1.64***	0.03
Number of relationships (*R*)	0.077***	0.20
Number of operations in the classes (*NumOpsCls*)[a]	0.029***	0.30
Number of classes in sub-packages (*NumCls_tc*)[a]	0.006***	0.00

(R^2) equals 0.81 with an intercept of 3.09.
$^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$.
[a]Metric used as a control variable in the regression model.

Table 11
Spearman's correlation between package size and maintainability metrics.

Metrics	Correlation
Package size (*NumCls*) and efferent coupling (*Ce*)	0.57***
Package size (*NumCls*) and afferent coupling (*Ca*)	0.39***
Package size (*NumCls*) and connected components (*ConnComp*)	0.69***
Package size (*NumCls*) and relational cohesion (*H*)	0.14***
Package size (*NumCls*) and number of relationships (*R*)	0.70***

$^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$.

Table 12
Summary of the relationship results of package size metric with maintainability metrics.

Analysis method	*Ce*	*Ca*	*ConnComp*	*H*	*R*
Correlation analysis	Moderate	Weak	Strong	Negligible	Strong
Regression analysis	Weak	Weak	Strong	Strong (−ve)	Weak
Effect size	None	None	Medium	Small	Medium

4.2.3. Size and complexity

Finally, the Spearman's correlation value suggests a strong relationship between package size and complexity metric (*R*)—see Table 11. However, it is not confirmed by the regression analysis as the coefficient value of complexity metric (*R*) is small; still, its effect size is medium (see Table 10).

> For refactoring large packages, package cohesion (measured through *ConnComp* metric) and complexity (computed using *R* metric) characteristics could be used as they have shown significant relationships with package size.

5. Discussion and validity threats

This section discusses the results and threats to validity of this study.

5.1. Discussion

Refactoring large packages solely based on size (i.e., number of classes—*NumCls*) may not always result in an optimal package structure. Even though our results show that large packages are more likely to have maintainability issues, it may be possible that not all large packages have maintainability issues as measured by the metrics in our analysis. For instance, a large package will not have cohesion issues if all the classes in it (even too many) are cohesive (i.e., all the classes forming only one connected component—*ConnComp*). In our dataset, we found 42 such instances of large packages, where the packages have only one connected component (*ConnComp*); for example, *org.apache.batik.anim.values*, *org.apache.cassandra.db.marshal*, and *org.apache.cayenne.merge*. This analysis also hints that there could be other large packages that may have good coupling or complexity. This means that metrics, besides size-related, also matter for large package refactoring, and the assumption of refactoring packages solely because they are large may not be always beneficial. In this section, we discuss the maintainability metrics (identified by our empirical analysis) that can help identify refactoring opportunities for large packages.

Our empirical study identifies the maintainability metrics useful for refactoring large packages and aligns with the prior arguments made in the literature on the importance of large package modularization. Although it is a common understanding that maintainability characteristics can be used for optimizing the overall package structure, there was a lack of empirical evidence on identifying the specific metrics useful for optimizing the package size. In addition, as also revealed by our results, not all maintainability metrics can be used for refactoring large packages. For instance, our results bring empirical evidence that maintainability metrics (related to cohesion and complexity) can be used for identifying refactoring opportunities for large packages; however, coupling metrics (*Ce* and *Ca*) may not be beneficial. Our identification of the metrics (beneficial for refactoring) is based on their strong association with package size and considerable effect size; for instance, package cohesion (*ConnComp*) and complexity (*R*) metrics.

Refactoring of large packages also aligns with the prior arguments made in the literature on why the modularization of large packages is beneficial for independent software development and effective communication among developers (Martin, 2002). In the literature, it is discussed that large packages can discourage the development team from working independently (Martin, 2002). In addition, they can negatively impact the communication and coordination in the development team because if many developers are working on a single package, there might be communication hindrances and coordination issues among them (i.e., lacking communicability and socio-technical congruence) (Martin, 2002). The ineffective communication and coordination in the development teams can introduce sub-optimal socio-technical patterns in the organizational structure commonly known as "community or social smells" (Tamburri et al., 2019). For example, if multiple developers are working on the smaller packages (that are dependent on a larger package), the communication between the developers might create a bottleneck effect (*radio silence* community smell Tamburri et al., 2019) because all the communication passes through the developer(s) working on the large package.

Refactoring large packages based on cohesion metric (*ConnComp*). Based on the results, if a large package needs refactoring, more focus should be given on improving the package cohesion by reducing the connected components (measured by *ConnComp* metric) because a low number of connected components indicates good cohesion. We discuss this cohesion-based refactoring strategy that has been previously applied by Lippert and Roock (2006) and Palomba et al. (2015). The refactoring is reducing the connected components by splitting the package into multiple packages (Lippert and Roock, 2006; Palomba et al., 2015). For instance, if a package has three connected components (represented by the groups of connected classes), each component can be extracted into a package, forming three packages. Then, each package would only contain the coupled classes (i.e., classes communicating with each other), fulfilling the single responsibility principle. As a result, the packages would be better semantically structured with improved package sizes. When a package has ideally only one connected component or less connected components (in general), it is an indication that package size has improved in terms of its cohesion. Software practitioners can benefit from this result by keeping the number of connected components (*ConnComp*) to a minimum—less connected components (*ConnComp*) means they can maintain better cohesion, and can eliminate unnecessary classes in the package (i.e., improved package size). The practical usefulness of this result can be amplified if tool developers can include the visualization of the connected components using graphs to ease the identification of the clusters of classes that are not communicating with each other. Once such clusters are visually identified, the developers can refactor by splitting the package into multiple packages, where each package contains a cluster. Potentially, refactoring can also be enabled visually in the tools.

Refactoring large packages based on complexity metric (*R*). Based on the strong association of the complexity metric (*R*) with package size, it can also be used for identifying refactoring needs of large packages. Again, the refactoring strategy is adopted from the existing literature (Nongpong, 2015; Suryanarayana et al., 2015). The complexity-based refactoring would likely need to occur in two phases because the increase in the complexity (number of relationships (*R*)) indicates that coupling between classes of the packages is also increasing (i.e., "feature envy" design smell) (Nongpong, 2015). Therefore, first, "move method" refactoring should be applied to reduce the method calls (relationships) between the classes or interfaces of a package (i.e., removing the "feature envy" smell) (Suryanarayana et al., 2015). With "move method" refactoring, each class will be responsible for executing a particular functionality (i.e., expressing the single responsibility principle). Second, "extract class" refactoring can be applied to move the uncoupled (non-cohesive) classes to a new package resulting in reduced package sizes (Suryanarayana et al., 2015). Once the number of relationships (*R*) in a package is in a manageable state (i.e., not many), it is an indication that the package size is optimized in terms of its complexity. Again, the complexity-based refactoring can help practitioners perform informed and timely refactoring of large packages so that they can maintain package sizes. Furthermore, tool builders can incorporate the complexity metric (number of relationships—*R*) in their tools to assist their users in applying the "move method" and "extract class" refactoring strategies.

So far in this discussion, we have demonstrated how the two strongly correlated metrics can be used for refactoring large packages. However, future work could include other package metrics (measuring coupling, cohesion, and complexity) to gain additional insights into their usefulness for package size optimization. In addition, literature has emphasized on modularized packages for improved socio-technical congruence and communicability in the software projects. Therefore, future work could also examine how the large packages impact the socio-technical aspects of the software projects. Lastly, this study only focuses on identifying refactorings for large packages; future work could investigate the metrics useful for refactoring small packages.

Another important aspect is the trade-off between modular disruption and package quality improvement that could be considered while refactoring large packages. Modular disruption considers the number of elements or modules that need to be moved or merged (Paixao et al., 2017) (i.e., the more the movement of classes between packages, the more disruption happens in the overall package structure). Therefore, developers need to decide whether they are willing to undertake the disruption of architecture (as a result of refactoring large packages) in comparison to the quality gains. By utilizing appropriate metrics to identify maintainability issues in package structure, some disruption will likely occur. However, over time, the disruption is likely to be reduced since less changes will be required to package structure after its initial optimization. Future work should empirically validate the disruptions and improvements as the software projects evolve over time. In addition, future work can apply methods (e.g., multi-objective search-based approach Paixao et al., 2017) to help developers find a balance between disruption and package improvement caused by refactoring.

Refactoring large packages based on coupling metrics may not be beneficial. Based on our results, we found that the coupling metrics (*Ce* and *Ca*) may not be helpful in optimizing the package size. The rationale for this lies in the refactoring strategy of large packages based on coupling. Usually coupling of packages is improved by "moving classes" between packages. This means that, on one side, where the package size is reduced by moving the classes, on the other side, the sizes of other packages (where the classes are moved) are increasing. Therefore, the overall effect of the change in the package structure (size-wise) is not that impactful; however, the coupling is improved. The refactoring strategies, based on our cohesion (*ConnComp*) and complexity (*R*) metrics, rely on "extracting classes" to new packages, resulting in overall modularized package structure.

Our regression model suggests that large packages may not suffer from additional coupling; however, based on the raw values of efferent (*Ce*) and afferent couplings (*Ca*), our findings also suggest that large packages may be more unstable. This indicates that dependencies with unstable large packages may lead to more instability in the package structure because instability is the ratio of efferent coupling (Ce) to total coupling ($Ce + Ca$) (Martin, 2002). When efferent coupling (*Ce*) is greater than afferent coupling (*Ca*), the instability will be higher in the package. Ideally, the instability value should be close to zero, indicating a stable package (Martin, 2002). A completely unstable package has an instability value of 1 (Martin, 2002). Our results indicate that, as the package size grows, the number of external classes or interfaces that the package depends on (i.e., efferent coupling (*Ce*)) increases at a greater rate than the number of external classes or interfaces that depend on the package (i.e., afferent coupling (*Ca*)). The risk ratios (in Table 9) also indicate this behavior as we notice large packages having more likelihood to suffer from efferent coupling (*Ce*) issues than afferent coupling (*Ca*). By performing additional analysis, we found that nearly half of the large packages (based on more than 22 classes) has instability value 0.6 or more, suggesting that efferent coupling (*Ce*) issue is more evident than afferent coupling (*Ca*) issue in large packages. To give a specific example, in Apache Ant project, *org.apache.tools.ant.filters* package is a large one with 32 classes and has an instability value of 0.818, meaning the package is highly unstable. This is happening because the package has efferent coupling (*Ce*) value of 27 which is significantly higher than afferent coupling (*Ca*) value of 6. Similarly, we observe a likewise behavior in another large package in Hibernate project. We notice a high efferent coupling (*Ce*) value of 60 in *org.hibernate.event.internal* package which made this package an unstable one (instability value is 0.895). This additional in-depth analysis suggests that large packages are more unstable. Ideally, the package structure should have a mix of stable and unstable packages because, in case all the packages are stable, the system will be rigid (unchangeable) (Martin, 2002). Therefore, the plan to action could be that the package structure should have high-level architecture and design decisions into stable packages (efferent coupling (*Ce*) will be minimal in stable packages), while, volatile functionality should be placed into unstable packages (the packages with minimal afferent coupling (*Ca*)).

Empirical analysis is needed to identify appropriate thresholds for the maintainability metrics that are strongly associated with packages size to assist refactoring decisions. Large packages can be refactored to the degree where maintainability metrics fall into good or, at most, regular (acceptable) ranges. In the literature, we could only find the threshold ranges of the coupling metrics (efferent (*Ce*) and afferent (*Ca*)) (Filó et al., 2015). However, our results show that these coupling metrics have zero effect size in relation to the package size; therefore, they are not useful for identifying the refactoring opportunities in large packages. Future work could conduct empirical studies to determine threshold ranges (good, regular, and bad) of the maintainability metrics that are useful for package size maintenance (e.g., cohesion—*ConnComp* and complexity—*R*). Future work could also focus on integrating these maintainability metrics into the development tools that will assist the developers in choosing the packages that they should consider refactoring.

5.2. Threats to validity

We explain the validity threats of our empirical study in terms of following four categories (as presented by Wohlin et al. (2003)).

5.2.1. Conclusion validity

This validity threat refers to the ability to draw correct conclusions regarding the relationship between the dependent and independent variables (Wohlin et al., 2003). In our study, a threat to conclusion validity is related to the large package classification. We considered large packages as those with 22 or more classes, which is the largest 10% of packages in our dataset. We do not claim this to be an exact threshold because packages that are close in size to 22 (e.g., 21 or 20 classes) may also be considered large packages. However, since our goal was to compare large packages with other packages, some cut off was needed. We have shown there are differences in large packages using 22 classes as a cut off in the set of analyzed projects. However, other threshold value for large packages (or package classification in general) may produce different outcome. The statistical analysis also showed significant differences in package sizes, indicating the validity of the employed thresholds. However, future studies can investigate whether similar large package issues exist in slightly smaller packages. Lastly, despite employing well-known statistical methods in our analysis, there could still be a possibility that the relationships identified through the methods are prone to biasness.

5.2.2. Construct validity

The construct validity deals with the accurate representation of the theoretical concepts in the dependent and independent variables (Wohlin et al., 2003). To mitigate this, we used a set of metrics that are the most commonly studied in the literature. For instance, the coupling metrics (efferent (*Ce*) and afferent(*Ca*)) have been widely employed in many studies to measure the coupling attribute (Abdeen et al., 2009; Ampatzoglou et al., 2019; Almugrin et al., 2016; Chantian and Muenchaisri, 2019). Similarly, other considered metrics have been widely used. However, the use of other metrics in the analysis may yield additional insights into the relationship between package size and its maintainability.

Another construct validity threat is related to the metric (number of classes—*NumCls*) used for measuring the package size. This metric only counts the number of classes that are directly in the package (i.e., excluding the classes in sub-packages). This conceptualization of package size is commonly used in the literature (Lippert and Roock, 2006; Chantian and Muenchaisri, 2019). Therefore, we also employed the package size metric that gives the count of classes that are directly in the package. However, it can be argued that the classes in sub-packages might be related to the main package. Therefore, future work could examine how the sizes of sub-packages impact the maintainability of packages. Lastly, the properties (related to coupling, cohesion, and complexity) of packages may vary depending on the project type. For instance, when packages are structured based on features, strong cohesion would be expected; whereas, when packages represent a service layer, lower cohesion would be expected. Future work could investigate the optimal package structures for different project types.

5.2.3. Internal validity

This validity threat means that the changes in the dependent variable are caused by the changes in the independent variable (Wohlin et al., 2003). In our empirical study, an internal validity threat is related to the imprecise measurement of the variables because if variables are measured imprecisely, there could be error-in-variable bias. To mitigate this, we automated the measurement of the metrics using well-known tools. Another internal threat is related to the collinearity between the independent variables. To mitigate this, we measured the Variance Inflation Factor (VIF) of the maintainability metrics to ensure no multicollinearity exists between them.

5.2.4. External validity

The external validity is related to the generalization of the experimental results (Wohlin et al., 2003). The quantity of projects is usually a common threat to external validity. To mitigate this, we employed a large set of open-source projects belonging to a well-known set of Java projects compiled in *Qualitas Corpus*. The *Qualitas Corpus* have different types of Java projects (e.g., analysis tools, management tools, parsers, and drivers), making the corpus appropriate representative of the Java projects (Terra et al., 2013; Tempero et al., 2010). However, we cannot claim that our results can generalize beyond our dataset. The inclusion of additional projects can examine the generalization of the results. In addition, we considered only open-source Java projects; therefore, the inclusion of commercial-based Java projects may produce additional insights (from the industry perspective). Furthermore, our focus was on the analysis of Java package structure; therefore, we only considered Java-based projects. Software projects developed using other languages (e.g., C# and C++) can verify whether our results generalize to other software ecosystems.

6. Conclusion

In this paper, we analyzed the package maintainability issues (by employing the package-level metrics) associated with different package sizes and identified potential refactoring opportunities for large packages. We performed the analysis by investigating the relationships between package size and three internal maintainability characteristics (coupling, cohesion, and complexity) using the package-level metrics collected from a set of 111 open-source Java projects compiled in the *Qualitas Corpus*.

Our analysis of the maintainability metrics showed that large packages are more vulnerable to maintainability issues in comparison with moderate and small packages. In addition, based on the coupling metrics, we reported that large packages can be more unstable. These results suggest the importance of refactoring large packages. Based on the relationships and effect sizes, we found that some maintainability metrics can be more useful for identifying refactoring opportunities for large packages. For instance, we showed that the strong correlations and moderate effect sizes of cohesion (*ConnComp*) and complexity (*R*) metrics with package size can be used for refactoring large packages. These maintainability metrics could also be integrated into development tools to enable automated and visual refactoring.

Future work could include other metrics related to coupling, cohesion, and complexity to gain additional insights into their relationships with package size. In addition, development tools can integrate the strongly correlated metrics to assist the developers in identifying when the large packages should be refactored. Lastly, we recommend investigating the relationship of large packages with the socio-technical issues in the software projects.

CRediT authorship contribution statement

Haris Mumtaz: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Paramvir Singh:** Conceptualization, Methodology, Writing – review & editing, Supervision. **Kelly Blincoe:** Conceptualization, Methodology, Writing – review & editing , Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the data and code files using Figshare platform. The link to the data/code is embedded in the manuscript.

References

Abdeen, H., Ducasse, S., Sahraoui, H., Alloui, I., 2009. Automatic package coupling and cycle minimization. In: Proceedings of the 16th Working Conference on Reverse Engineering. IEEE, pp. 103–112. http://dx.doi.org/10.1109/WCRE.2009.13.

e Abreu, F.B., Goulao, M., 2001. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In: Proceedings of the 5th European Conference on Software Maintenance and Reengineering. IEEE, pp. 47–57. http://dx.doi.org/10.1109/CSMR.2001.914968.

Aggarwal, K., Singh, Y., Kaur, A., Malhotra, R., 2008. Application of artificial neural network for predicting maintainability using object-oriented metrics. Int. J. Comput. Electr. Autom. Control Inf. Eng. 2 (10), 3552–3556.

Al Dallal, J., 2013. Object-oriented class maintainability prediction using internal quality attributes. Inf. Softw. Technol. 55 (11), 2028–2048. http://dx.doi.org/10.1016/j.infsof.2013.07.005.

Almugrin, S., Albattah, W., Melton, A., 2016. Using indirect coupling metrics to predict package maintainability and testability. J. Syst. Softw. 121, 298–310. http://dx.doi.org/10.1016/j.jss.2016.02.024.

Alves, T.L., Ypma, C., Visser, J., 2010. Deriving metric thresholds from benchmark data. In: Proceedings of the IEEE International Conference on Software Maintenance. IEEE, pp. 1–10. http://dx.doi.org/10.1109/ICSM.2010.5609747.

Ampatzoglou, A., Tsintzira, A.-A., Arvanitou, E.-M., Chatzigeorgiou, A., Stamelos, I., Moga, A., Heb, R., Matei, O., Tsiridis, N., Kehagias, D., 2019. Applying the single responsibility principle in industry: Modularity benefits and trade-offs. In: Proceedings of the Evaluation and Assessment on Software Engineering. ACM, pp. 347–352. http://dx.doi.org/10.1145/3319008.3320125.

Anacleto, V.A., 2008. A UML profile for documenting the component-and-connector view of software architectures. J. Comput. Sci. Technol. 8 (1), 21–26.

Aniche, M., Treude, C., Zaidman, A., Van Deursen, A., Gerosa, M.A., 2016. SATT: Tailoring code metric thresholds for different software architectures. In: Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 41–50. http://dx.doi.org/10.1109/SCAM.2016.19.

Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. 28 (1), 4–17.

Beck, K., Fowler, M., Beck, G., 1999. Bad smells in code. Refactoring: Improv. Des. Exist. Code 1 (1999), 75–88.

Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E., 2008. Fast unfolding of communities in large networks. J. Statist. Mech.: Theory Exp. 2008 (10), P10008.

Briand, L.C., Daly, J., Porter, V., Wust, J., 1998. A comprehensive empirical validation of design measures for object-oriented systems. In: Proceedings of the 5th International Software Metrics Symposium. IEEE, pp. 246–257. http://dx.doi.org/10.1109/METRIC.1998.731251.

Briand, L.C., Morasca, S., Basili, V.R., 1993. Measuring and assessing maintainability at the end of high level design. In: Conference on Software Maintenance. IEEE, http://dx.doi.org/10.1109/ICSM.1993.366952.

Briand, L.C., Wüst, J., 2002. Empirical studies of quality models in object-oriented systems. Adv. Comput. 56, 97–166. http://dx.doi.org/10.1016/S0065-2458(02)80005-5.

Chantian, B., Muenchaisri, P., 2019. A refactoring approach for too large packages using community detection and dependency-based impacts. In: Proceedings of the World Symposium on Software Engineering. ACM, pp. 27–31. http://dx.doi.org/10.1145/3362125.3362132.

Cohen, J., 2013. Statistical Power Analysis for the Behavioral Sciences. Academic Press, http://dx.doi.org/10.4324/9780203771587.

Dagpinar, M., Jahnke, J.H., 2003. Predicting maintainability with object-oriented metrics-an empirical comparison. In: Proceedings of the 10th Working Conference on Reverse Engineering. IEEE Computer Society, p. 155. http://dx.doi.org/10.1109/WCRE.2003.1287246.

Elish, M.O., Elish, K.O., 2009. Application of TreeNet in predicting object-oriented software maintainability: A comparative study. In: 13th European Conference on Software Maintenance and Reengineering. IEEE, pp. 69–78. http://dx.doi.org/10.1109/CSMR.2009.57.

Falessi, D., Kazman, R., 2021. Worst smells and their worst reasons. In: IEEE/ACM International Conference on Technical Debt. pp. 45–54. http://dx.doi.org/10.1109/TechDebt52882.2021.00014.

Falk, R.F., Miller, N.B., 1992. A Primer for Soft Modeling. University of Akron Press.

Ferreira, K.A., Bigonha, M.A., Bigonha, R.S., Mendes, L.F., Almeida, H.C., 2012. Identifying thresholds for object-oriented software metrics. J. Syst. Softw. 85 (2), 244–257. http://dx.doi.org/10.1016/j.jss.2011.05.044.

Filó, T.G., Bigonha, M., Ferreira, K., 2015. A catalogue of thresholds for object-oriented software metrics. In: Proceedings of the 1st International Conference on Advances and Trends in Software Engineering. pp. 48–55.

Fontana, F.A., Ferme, V., Zanoni, M., Yamashita, A., 2015. Automatic metric thresholds derivation for code smell detection. In: Proceedings of the IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics. IEEE, pp. 44–53. http://dx.doi.org/10.1109/WETSoM.2015.14.

Harman, M., Hierons, R.M., Proctor, M., 2002. A new representation and crossover operator for search-based optimization of software modularization. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, Vol. 2. pp. 1351–1358. http://dx.doi.org/10.1.1.144.5252.

Henry, S., Kafura, D., 1981. Software structure metrics based on information flow. IEEE Trans. Softw. Eng. (5), 510–518.

Lanza, M., Marinescu, R., 2007. Object-Oriented Metrics in Practice: using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer Science & Business Media, http://dx.doi.org/10.5555/1965070.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018. An empirical study of architectural decay in open-source software. In: Proceedings of the IEEE International Conference on Software Architecture. IEEE, pp. 176–17609. http://dx.doi.org/10.1109/ICSA.2018.00027.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23 (2), 111–122. http://dx.doi.org/10.1016/0164-1212(93)90077-B.

Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. John Wiley & Sons.

Lorenz, M., Kidd, J., 1994. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, Inc., http://dx.doi.org/10.5555/177063.

Malakuti, S., Heuschkel, J., 2021. The need for holistic technical debt management across the value stream: Lessons learnt and open challenges. In: /ACM International Conference on Technical Debt. pp. 109–113. http://dx.doi.org/10.1109/TechDebt52882.2021.00021.

Martin, R.C., 2002. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

Matthes, F., Buckl, S., Leitel, J., Schweda, C.M., 2008. Enterprise Architecture Management Tool Survey. Technische Universität München.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A., 2015. Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Methodol. 24 (3), 1–45.

Mori, A., Vale, G., Viggiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., Kastner, C., 2018. Evaluating domain-specific metric thresholds: An empirical study. In: Proceedings of the IEEE/ACM International Conference on Technical Debt. IEEE, pp. 41–50. http://dx.doi.org/10.1145/3194164.3194173.

Nongpong, K., 2015. Feature envy factor: A metric for automatic feature envy detection. In: Proceedings of the 7th International Conference on Knowledge and Smart Technology. IEEE, pp. 7–12. http://dx.doi.org/10.1109/KST.2015.7051460.

Paixao, M., Harman, M., Zhang, Y., Yu, Y., 2017. An empirical study of cohesion and coupling: Balancing optimization and disruption. IEEE Trans. Evol. Comput. 22 (3), 394–414. http://dx.doi.org/10.1109/TEVC.2017.2691281.

Palomba, F., Tufano, M., Bavota, G., Oliveto, R., Marcus, A., Poshyvanyk, D., De Lucia, A., 2015. Extract package refactoring in aries. In: Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 669–672.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. In: Pioneers and their Contributions to Software Engineering. Springer, pp. 479–498. http://dx.doi.org/10.1145/361598.361623.

Prion, S., Haerling, K.A., 2014. Making sense of methods and measurement: Spearman–Rho ranked-order correlation coefficient. Clin. Simul. Nurs. 10 (10), 535–536. http://dx.doi.org/10.1016/j.ecns.2014.07.005.

Rizvi, S., Khan, R.A., 2010. Maintainability estimation model for object-oriented software in design phase (MEMOOD). J. Comput. 2 (4), 26–32.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131–164.

Sales, V., Terra, R., Miranda, L.F., Valente, M.T., 2013. Recommending move method refactorings using dependency sets. In: Proceedings of the 20th Working Conference on Reverse Engineering. IEEE, pp. 232–241. http://dx.doi.org/10.1109/WCRE.2013.6671298.

Samli, R., Aydın, Z.B.G., Yücel, U.O., 2020. Measurement in software engineering: The importance of software metrics. In: Applications and Approaches to Object-Oriented Software Design: Emerging Research and Opportunities. IGI Global, pp. 166–182. http://dx.doi.org/10.4018/978-1-7998-2142-7.ch007.

Selya, A.S., Rose, J.S., Dierker, L.C., Hedeker, D., Mermelstein, R.J., 2012. A practical guide to calculating Cohen's f^2, a measure of local effect size, from PROC MIXED. Front. Psychol. 3, 111. http://dx.doi.org/10.3389/fpsyg.2012.00111.

Seng, O., Bauer, M., Biehl, M., Pache, G., 2005. Search-based improvement of subsystem decompositions. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation. pp. 1045–1051. http://dx.doi.org/10.1145/1068009.1068186.

Sousa, B.L., Bigonha, M.A., Ferreira, K.A., Gerais, M., 2017. A Tool for Detection of Co-Occurrences Between Design Patterns and Bad Smells. Tech. Rep., Programming Language Lab.

Sullivan, G.M., Feinn, R., 2012. Using effect size—or why the P value is not enough. J. Grad. Med. Educ. 4 (3), 279–282. http://dx.doi.org/10.4300/JGME-D-12-00156.1.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2015. Refactoring for software design smells. ACM SIGSOFT Softw. Eng. Not. 40, http://dx.doi.org/10.1145/2830719.2830739.

Tamburri, D.A., Palomba, F., Kazman, R., 2019. Exploring community smells in open-source: An automated approach. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2019.2901490.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010. The qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference. IEEE, pp. 336–345. http://dx.doi.org/10.1109/APSEC.2010.46.

Terra, R., Miranda, L.F., Valente, M.T., Bigonha, R.S., 2013. Qualitas.class corpus: A compiled version of the qualitas corpus. ACM SIGSOFT Softw. Eng. Not. 38 (5), 1–4. http://dx.doi.org/10.1145/2507288.2507314.

Vale, G., Fernandes, E., Figueiredo, E., 2019. On the proposal and evaluation of a benchmark-based threshold derivation method. Softw. Qual. J. 27 (1), 275–306. http://dx.doi.org/10.1007/s11219-018-9405-y.

Wohlin, C., Höst, M., Henningsson, K., 2003. Empirical research methods in software engineering. In: Empirical Methods and Studies in Software Engineering. Springer, pp. 7–23. http://dx.doi.org/10.1007/978-3-540-45143-3_2.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

Zhang, J., Kai, F.Y., 1998. What's the relative risk?: A method of correcting the odds ratio in cohort studies of common outcomes. JAMA 280 (19), 1690–1691. http://dx.doi.org/10.1001/jama.280.19.1690.

Zhou, Y., Leung, H., 2007. Predicting object-oriented software maintainability using multivariate adaptive regression splines. J. Syst. Softw. 80 (8), 1349–1361. http://dx.doi.org/10.1016/j.jss.2006.10.049.

Haris Mumtaz is currently a Ph.D. candidate at the Department of Electrical, Computer, and Software Engineering of the University of Auckland, New Zealand. He acquired his master's degree in software engineering from the King Fahd University of Petroleum and Minerals, Saudi Arabia in 2016. His research interests include software quality, antipatterns, software metrics, software analysis, software visualization, and empirical software engineering.

Paramvir Singh is a Professional Teaching Fellow at the School of Computer Science of the University of Auckland, New Zealand. Prior to this, he worked as an Assistant Professor at the National Institute of Technology, India. His research interests include software design and architecture, evidence-based software engineering, dynamic program analysis, and music technology.

Kelly Blincoe is a Senior Lecturer of Software Engineering at the University of Auckland, New Zealand in the Department of Electrical, Computer, and Software Engineering. She leads the Human Aspects of Software Engineering Lab (HASEL). Her research is mainly in the human and social aspects of software engineering, including software dependencies, software ecosystems, collaborative software development, software requirements engineering, and software developer diversity and inclusion. Her research is currently funded by a Royal Society Rutherford Discovery Fellowship, the Royal Society Marsden Fund, the National Science Challenges Science for Technological Innovation's Veracity Spearhead, and Google.