The Journal of Systems and Software 164 (2020) 110543

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Lexical content as a cooperation aide: A study based on Java software

Andrea Capiluppi[a,*], Nemitari Ajienka[b]

[a] Dept. of Computer Science, Brunel University London, UK
[b] Dept. of Computer Science, Edge Hill University, UK

ARTICLE INFO

Article history:
Received 9 September 2019
Revised 5 December 2019
Accepted 7 February 2020
Available online 11 February 2020

Keywords:
Information retrieval (IR)
Lexical content
Clustering
Distributed development
Object-oriented (OO)
Open-source software

ABSTRACT

Collaborative development is a paradigm shift in software development. Loosely coupled developers coordinate their work via distributed versioning systems (SVN, Git, and others), code reviews and priority-led bug tracking systems. This development approach allows many different developers to input additional source code to the same source artifact.

This article focuses on the lexical content of the source code produced in a collaborative environment. The lexical content is described as the 'dictionary' of the key terms contained within a source artifact. We posit that the lexical content of a Java class will increase as long as more developers add more content to the same class.

We analyse the 100 top-ranked GitHub applications (at the time of the sampling) written in Java. Each of their classes is reduced to its lexical content, its size (in LOCs) recorded, as well as the number of different developers who contributed to its source code.

Our results show that (i) the lexical content of Java classes is bounded in size, (ii) more developers make the size of the lexical content larger, and (iii) the lexical content of a system's classes might increase with more developers, but depending on its application domain.

The implications for practitioners are two-fold: (i) classes with a large set of lexical content should be split in multiple classes, to minimize the need for further maintenance; and (ii) classes developed by many developers should adhere to specific guidelines so that its lexical content does not increase boundlessly. We tested our results in a tailored case study and we confirmed our findings: larger-than-threshold class corpora tend to deteriorate the class cohesion.

1. Introduction

Global software development has long been recognised as a paradigm shift in modern software development (Kogut and Metiu, 2001; Ebert and De Neve, 2001; Carmel and Agarwal, 2001). As an immediate effect, co-location of workers in the same building or office, deemed as necessary and unique (Olson et al., 2002), is not seen as necessary any longer (Bird and Nagappan, 2012). Coordination in distributed socio-technical systems is mostly achieved by means of the artifacts that are produced by the developers part of a project's team (Scholtes et al., 2016; Roberts et al., 2006; Bird, 2011).

With communication becoming less frequent, the challenge is for it to become more effective. This is especially complex when different nationalities, languages and cultures are part of the same development effort (Richardson et al., 2012; Noll et al., 2010; Verner et al., 2014). It has been demonstrated that the absence of a shared native language (known as "linguistic distance") creates further barriers to communication (Herbsleb, 2007; Carmel and Tjia, 2005).

Open source software is an example of a distributed, multilingual development effort (Gonzalez-Barahona et al., 2016; Devanbu et al., 2017; Capiluppi and Ajienka, 2018). As such, the main resulting artefacts are online discussions, and source code. Software developers from diverse ethnic backgrounds and language groups usually posses different coding techniques and approaches (Middleton, 2001; Noll et al., 2010), and could develop software artefacts with corpora at varying levels of semantic or lexical richness. For example, in the source code comments or feature identifiers.

This variety could have diverging effects on the ability to collaborate between developers: if the underlying classes are large and

* Corresponding author.
E-mail addresses: andrea.capiluppi@brunel.ac.uk (A. Capiluppi), nemitari.ajienka@edgehill.ac.uk (N. Ajienka).

https://doi.org/10.1016/j.jss.2020.110543
0164-1212/

lexically complex, collaboration between diverse developers would be hindered due to complexity in program comprehension (Rajlich and Wilde, 2002; Wettel and Lanza, 2007). On the other hand, lexically simpler classes, although as large, would not be problematic to collaborate on. This is because the content of a class lexicon is partly covered by the classic Halstead metric (Halstead et al., 1977), that has been linked with improved readability in various case studies (Posnett et al., 2011; Daka et al., 2015).

In a motivating study, Host and Ostvold (Host and Ostvold, 2007) emphasise that method identifiers make or break abstractions: while good identifiers communicate the intention of the method, bad ones tend to cause confusion and frustration (Niu et al., 2012). Furthermore, the task of creating identifiers is subject to the sudden ideas and way of thinking of the individual as programmers have little to guide them except their personal experience. Based on this idea, Host and Ostvold analysed method implementations taken from a corpus of Java projects, and established the meaning of verbs in method identifiers based on actual use. As a result, they produced an automatically generated, domain-neutral lexicon of verbs, similar to a natural language dictionary, that represents the common usages of many programmers.

In this study, we systematically analyse the lexical complexity of Java classes, by parsing their source code, sifting through the language-specific terms, and keeping the terms that make up the lexicon of each Java class. We use a sample of 100 Java systems, selected from the most popular Java projects hosted on GitHub.com. The aim of this study is to evaluate the lexical content of Java classes, as factors for the collaboration between software developers.

This study is based on the following research questions:

RQ1 **Is the size of the lexical content an invariant among Java classes?**
Rationale: the lexical content of a class contains the unique terms that compose the dictionary of a class. For the purpose of maintainability and understandability, this content should be kept to a minimum, while at the same time be as different as possible from each other. This research question investigates whether the size of the set of such terms is invariant (or a quasi-constant) in a large and diverse collection of Java classes.

RQ2 **Is there a correlation between the unique developers collaborating on a class, and the size of its lexical content?**
Rationale: the more developers working on a class, the larger that class should become, and the more complicated the set of terms that new developers will add. It becomes important to learn how the basic dictionary of a class is affected by a stable (or growing) number of developers.

RQ3 **Is there a correlation between the size of the lexical content of a class, and its maintenance needs?**
Rationale: by maintenance needs we make reference to the number of changes or modifications made to a class (Al Dallal, 2013). Past literature has hinted to the relationship between size of a class and its complexity. This research question tries to establish whether the size of its lexical content is associated with a the number of changes performed on the same class.

The main contributions of this work are:

- we show the size distribution of the class corpora, in a large set of successful Java projects;
- we show how the lexical content of a class changes when more developers work on the same Java class;
- we investigate the link between the size of the lexical content of a Java class, and the likelihood that it is related to more or less changes;
- we evaluate the difference between test and non-test classes, and the influence of the application domains in the results that we collected;
- we focus on the experience of developers as a primary factor for the further maintenance of classes;
- we offer a qualitative case study that makes use of our findings to repair a few of the classes that show a larger-than-average corpus size.

This paper is articulated as follows: Section 2 describes the methodology of the empirical work, with definitions, research questions and how the attributes were operationalised for testing the hypotheses. Section 3 presents the results in the order of the research questions, while Section 4 discusses the relevance of these results in a wider context. Section 5 deals with the related work, in the context of open source and distributed collaboration, and the importance of lexical and semantic aspects within software development. Section 6 discusses the main threats to validity, while Section 7 offers our conclusion.

2. Empirical approach and methodology

The study presented here is based on the collection of Java classes, their lexical content (i.e., *corpus*) and the meta-data of which developers created or modified what classes in a system. The methodology of how to extract such data is explained below, together with a working example. The raw data and intermediate results are made available at https://figshare.com/projects/Lexicon_and_Developers_in_Java_classes/56009.

2.1. Definitions

This section defines the terminology as used throughout the paper. Each of the items described below will be operationalised in one of the subsections of the methodology.

- **Size** of a class – measuring the size of software systems has been a long tradition in empirical software engineering. The size of artefacts is always the first choice to define the extent of the functionalities contained in a system. In this paper we consider the size of a class considering both its physical lines of code (LOCs) and the source lines of code (SLOCs).
- **Corpus keyword** (term) – given the source code contained in a class, a term is any item that is contained in the source code. We do not consider as a term any of the Java-specific keywords (e.g., *if, then, switch*, etc.)[1] Additionally, the camelCase or PascalCase notations are first decoupled in their components (e.g., the class constructor *InvalidRequestTest* produces the terms *invalid, request* and *test*).
- **Class corpus** – with the term 'class corpus' we consider the set of all the terms contained within a class. We consider two types of class corpus, per class: the *complete* corpus, with all the terms contained; and the *unique* set of terms, that is, purged of the duplicates.
- **Committer** – all the projects in the case study presented below are taken from the GitHub.com online repository. The term '*committer*' applies to any developer of a project who has the right to commit the code onto the main trunk of the code base.
- **Author** – several developers are currently working on each of the projects in parallel. The mechanics of the *forking* feature

[1] The complete list of Java reserved words that we considered is available at https://en.wikipedia.org/wiki/List_of_Java_keywords. The `String` keyword was also considered as a reserved word, and excluded from the text parsing.

in Git facilitates the parallel development, and collaboration on different classes. We term as '*author*' any developer who contributes code to the project, although it might get committed to the main development trunk by another *committer* with the right write access. For the purpose of this paper, we have counted the number of *distinct* authors who have modified at some point any part of a Java class. Below, we consider *author* and *developer* as synonyms.

- **Developer cluster** – a Java class might get created and modified by one or many developers. A developer cluster is the set of all the Java classes developed by the same number of developers. For example, a developer cluster of 2 is the subset of all Java classes modified by exactly two developers.

2.2. Hypotheses

From the research questions described above, we formulate the following hypotheses:

$H_{0,1}$ the size of a class corpus is quasi-constant among classes, for both *complete* and *unique* corpora.
Test: this hypothesis will be tested by displaying the values of complete and unique class corpora in boxplot distributions.

$H_{0,2}$ the size of a class corpus does not increase as long as more developers contribute to the same Java class.
Test: this hypothesis is tested by plotting the average and median values of the class corpora, as clustered in developer *clusters*.

$H_{0,3}$ There is no correlation between the size of class corpora and number of changes of a class.
Test: this hypothesis is tested by running one Spearman test for the overall sample of Java classes, and several additional Spearman tests (one per project analysed). The variables tested are the size of class corpora, and the changes incurred by each class.

2.3. Dataset

In this study, we have investigated the link between the lexical richness of Java classes and collaboration in OO software. Leveraging the GitHub.com repository, we collected the project IDs of the 100 most successful Java projects hosted on GitHub.com as case studies.[2] The "success" of the projects is determined by the number of *stars* received by the community of GitHub users and developers, as a sign of appreciation. We used this approach to stratified sampling because the projects obtained by this filter are likely to be used by a large pool of users (Crowston et al., 2003), active in terms of the number of commits (Borges et al., 2015; Kalliamvakou et al., 2014) in the last 6 months preceding the sample collection for the study and potentially have a good intake of new developers (Von Krogh et al., 2003). Prior studies have also adopted similar selection criteria (Alshomali et al., 2017; Borges and Valente, 2019) when analysing software repositories hosted on GitHub.

Some 83,300 Java classes are contained in these projects, overall: the largest project of our sample is *elasticsearch*,[3] with over 9000 classes. The repository of each project was downloaded and stored, with its metadata (list of revisions for each class, and for the whole project, name of developers, date and time of changes), using the CVSAnalY set of tools[4,5]. These revisions do not contain files without the.java extension. A few of the studied projects have vast amounts of revisions (more than 10,000), but they represent the outliers of the distribution. In summary, the median of the number of revisions *per* project is 1000.

We extract the metadata of each Java class change, as stored on GitHub. Metadata comprises the unique class ID, the date and hour of each change on this class, the developer responsible for the change and the explanation of such change. Java classes can be developed by one or many developers, and on one or many parallel branches of development, as allowed by the Git technology.

This data extraction produces a list of classes and an associated number of distinct developers. Irrespective of the projects they come from, we group classes into '*clusters*' if they are developed by a similar number of developers, resulting in the one-developer *cluster*, two-developer *cluster* and so on.

2.4. Deriving the number of Java classes

The 100 selected systems are all written in Java, but the number of classes contained in each system varies according to the distribution shown in the boxplot[6] of Fig. 1. A small number of outlier contains a number of classes larger than 2000, but most systems are much smaller than that. The average number of classes in that set is 833, while the median of the set is 232 classes.

We mined the names of the classes (and their full path) to detect which subset composed the tests, and overall we found 31,400 classes that have "test" or "Test" in their name (or path). A distribution of these across the sample is given in the lower boxplot of Fig. 1.

Considering the systems in our sample, the amount of tests per system is highly correlated ($\rho = 0.78$) with the number of non-test classes. Larger systems tend to have a larger amount of tests: the subset of smaller-sized systems (i.e., less than 100 source classes), have an average 18% of tests as source code; systems between 100 and 1000 classes are composed of 34% of test classes (on average); while systems with over 1000 classes have on average 42% of test classes. These observations are important, since they will be used to determine how the lexical complexity affects the test classes, as compared to non-test classes.

2.5. Extracting the lexical content from Java classes

We extract the lexical content of a Java class in two ways: (i) by considering their class names; and (ii) parsing their code and considering all identifiers including method and variable names, comments and keywords.

The code of a Java class is converted into a *text corpus* where each line contains elements of the implementation of a class. The corpus in this case ("dictionary" of terms derived from comments, keywords in source code) is built at the *class* level of granularity (Kagdi et al., 2013).

The corpus includes the class name, variable and method names and comments for each class. Pre-processing of the system corpus is performed to eliminate Java keywords,[7] stop words, split and to stem class names (Marcus et al., 2004). The list of such terms is available in the replication package for inspection.

As per the definitions given above, for the analyses performed in this paper, we extracted both the *complete* and the *unique* corpus of each class. As an example, for the lines of code shown in Fig. 2 (the UrSQLEntry.java class from the UrSQL project), we derive the following *complete* corpus using an information retrieval tool developed in Java: {*ur sql entri kei valu kei valu ur sql entiti entiti*

[2] The list of projects is available at https://figshare.com/s/c627af8e9e496a9025c4.
[3] https://github.com/elastic/elasticsearch.
[4] http://metricsgrimoire.github.io/CVSAnalY/.
[5] Installation steps can be found at: https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/howtoinstallandruncvsanaly2inubuntu1110.
[6] All the boxplots shown in this paper have been produced using the online facility at http://shiny.chemgrid.org/boxplotr/.
[7] As shared on https://en.wikipedia.org/wiki/List_of_Java_keywords.

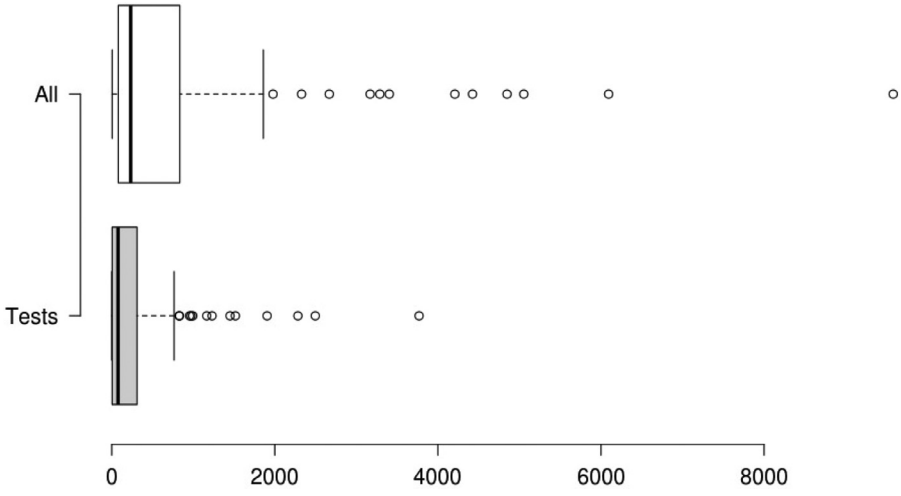

Fig. 1. Distribution of number of classes in the sample (including tests).

```
1  package tmacsoftware.ursql;
2
3  public class UrSQLEntry
4  {
5
6      private String key;
7      private String value;
8      private String firstKey;
9      private String firstValue;
10     private UrSQLEntity entity;
11
12     public UrSQLEntry()
13     {
14     }
15
16     public UrSQLEntry(String query)
17     {
18         String[] split = query.split
   (UrSQLController.KEY_VALUE_SEPARATOR);
19         this.key = split[0];
20         this.value = split[1];
21         this.firstKey = this.key;
22         this.firstValue = this.value;
23     }
```

Fig. 2. UrSQLEntry.java Source Code Snippet.

ur sql entri ur sql entri queri split queri split ur sql control kei valu separ kei split valu split kei kei valu valu}. The tool can be downloaded from Figshare,[8] and it uses the *ninka*[9] project to detect a source file's license, that is not considered relevant for a source file's lexicon.

In order to obtain the unique corpus, the list of keywords is later pruned of duplicated terms, per class. From Fig. 2, we derive the *unique* corpus as follows: {*control entiti entri kei queri separ split sql ur valu*}. The *complete* and the *unique* corpora are obviously different, the former being of size 35, and the latter of size 10 (in the example above).

As a summary, this data extraction produces, for each class, the relative size of both the complete and unique sets of its lexical corpus, which we also made available for inspection (and potential replication).

[8] https://figshare.com/articles/Script_to_extract_the_lexical_corpus_of_a_Java_class/9785861.

[9] Available at https://github.com/dmgerman/ninka, as presented in German et al. (2010).

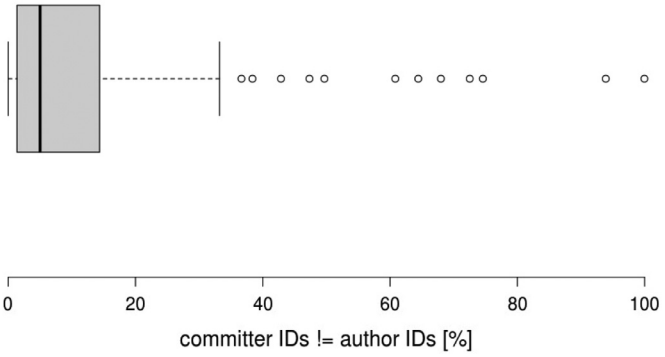

Fig. 3. Commits where committers are not the authors of the code contribution (per project, in percentage).

2.6. Identification of developers

In this section we show how we extracted the distinct authors responsible for modifications on the Java classes, and we consider them as the developers of the classes under study. The distinction between 'author' and 'committer' is quite crucial in our study, since we clustered Java classes into developer clusters (see Section 2.6.2): we show below how using 'committers' could skew the study.

In Fig. 3 below, we report on the difference between committers and authors, in terms of how atomic commits are acknowledged: work might get committed into the main developed trunk by one *committer*, but developed by a different *author* (i.e., the committer is not the author). For each commit involving a Java class, we extracted the committer and author IDs, and checked where the two refer to the same developer ID.[10] Fig. 3 shows the percentage of commits where the author ID is different from the committer ID, per project.

For the outlier projects of Fig. 3 (e.g., the *CoreNLP* project[11]), only a very limited number of developers are indeed committers.

[10] The SQL query used to identify the divergence is: `select files.repository_id, scmlog.author_id, scmlog.committer_id from scmlog, actions, files where actions.commit_id = scmlog.id and actions.file_id = files.id and scmlog.committer_id != scmlog.author_id and file_name LIKE '%.java%';`

[11] https://github.com/stanfordnlp/CoreNLP.

Table 1
Reconciliation of duplicate IDs in the developers metadata.

Project	Dev name	Dev ID	Reconciled dev ID
robolectric	petrcermak	11,136	11,015
robolectric	cermak	11,015	11,015
robolectric	Travis Collins	10,894	10,894
robolectric	travisc	11,097	10,894

On the other hand, over 90% of the *CoreNLP* commits are contributed by other developers, and acknowledged as authors in the commit message.

In other cases (left part of Fig. 3), such difference is less visible: most committers are indeed the authors of the code pushed onto the development trunk. The median value of commits where committers are not authors is 5% of the total commits, per project (average is 13%).

These results show, on the one hand, how the git *pull requests* (Kalliamvakou et al., 2014) work in practice: authors modify and contribute new code to a project, while being acknowledged by the core development team (committers). On the other hand, it is clear that all the empirical analyses performed below necessitate the use of authors as developers (instead of committers): author IDs produce a more complete picture of who modified the code of Java classes.

2.6.1. Removing duplicate authors

An important factor for the extraction of developer metadata is to avoid to include multiple times the same individuals. In this section we detail how this process was performed, in a semi-automatic way.

Names in the development log typically appear in three main forms:

1. in the '*Name Surname*' form (e.g., *Adam Smith*)
2. in the '*moniker*' form (e.g., *asmith*).
3. in the '*Name Surname* and *Name1 Surname1*' form, to acknowledge where two developers worked together (e.g., *Adam Smith and John M Keynes*).

In all the above cases, a distinct developer ID was automatically assigned in the database. The aim of this procedural step was to reconcile cases 1) and 2) onto the same developer ID; and to separate the two developers of case 3) while assigning new developer IDs.

In order to merge the cases 1) and 2), we isolated the *Surname* field of the former, and looked for the same pattern in the latter. In case that was found, the two developer IDs were merged (i.e., *reconciled*) into one. An example of this approach is shown in Table 1 below.

2.6.2. *Mechanics of developer* clustering

In this section we describe the approach we used to assign each class to a unique developer *cluster*: the goal was to separate classes developed by one developer from classes developed by two developers, three developers, and so on. As per the definitions above, we consider authors and developers as synonyms.

Fig. 4 illustrates two examples of data extraction for *presto*[12] and *zxing*[13]: in the *presto* project, the `StatisticsBenchmark` class has been modified by 3 developers, while `SlideReadFunction` and `TestWrappedRange` by one developer only. In the *zxing* project, class

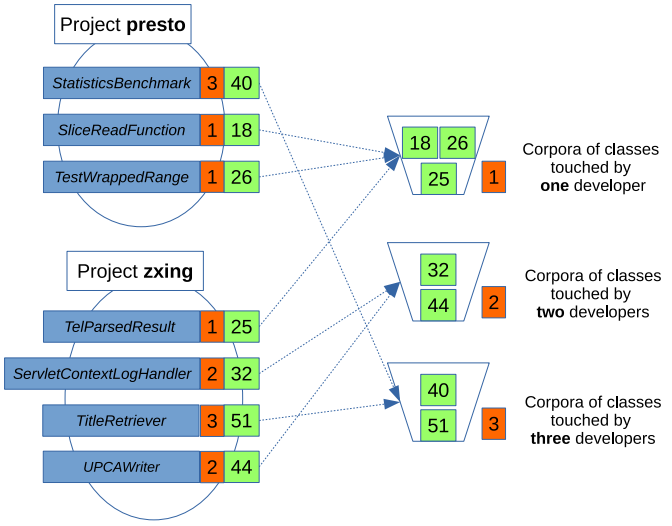

Fig. 4. Assignment of class corpora to developer *clusters*.

`TelParsedResult` has also been modified by only one developer, `ServletContextLogHandler` and `UPCAWriter` by two developers, and `TitleRetriever` by three developers.

We store the corpora of the `SlideReadFunction`, `TestWrappedRange` and `TelParsedResult` in the same *cluster* since they have the same number of developers (shown in the green colored squares beside each class); the same applies for classes `ServletContextLogHandler` and `UPCAWriter` whose corpora are stored in the two-developer *cluster*. Finally, the corpora of `StatisticsBenchmark` and `TitleRetriever` are stored in the three-developer *cluster*.

From the projects analysed, we observe that the size of these *clusters* is heavily biased: out of an overall 83,300 classes, there are some 15,000 classes (and corpora) that have been modified by only one developer; around 11,000 modified by 2 developers, and over 8700 modified by 3 developers. The next task is to compare these *clusters* in pairs, to detect: 1) if they come from the same population and 2) if they contain statistically different values.

3. Results

In this section we present the results of the empirical investigation, in the same order of the research questions. Different statistical tests were performed for the various RQs: we illustrate which test was used, and for each, how they were carried out, and the outcome of the tests.

3.1. RQ1 – analysis and results

The first research question is as follows: **Is the size of the lexical content an invariant among Java classes?** This research question ought to establish invariants in the size of the lexical corpora, for a large sample of Java classes. Following the definitions above, we extracted the *complete* and the *unique* corpus, per class, and for all the systems, and collected the size of these sets. For all the systems in the sample, we evaluated the median value of the two corpora: we prefer this metric to the average, given the skewness in the distribution of the corpora.

Overall sample As an overall sample, we observed that the corpus size in the pool has an average of 450 terms in the classes of the sample, but only a median of 179 terms. That by itself is a proof of how skewed is the data set. When considering the unique terms in the classes, we observed a much smaller value: as an

[12] https://github.com/prestodb/presto.
[13] https://github.com/zxing/zxing.

Table 2
Median, average and standard deviation (in number of terms) for the unique and complete corpora of Java classes, considering the overall sample.

	Unique corpus			Complete corpus		
	median	avg	std dev	median	avg	std dev
All classes	45	65	119.3	179	451	1493.4
Test classes	43	60	157.8	184	468.6	1263.31
All but test classes	47	68	88	176	440.61	1617

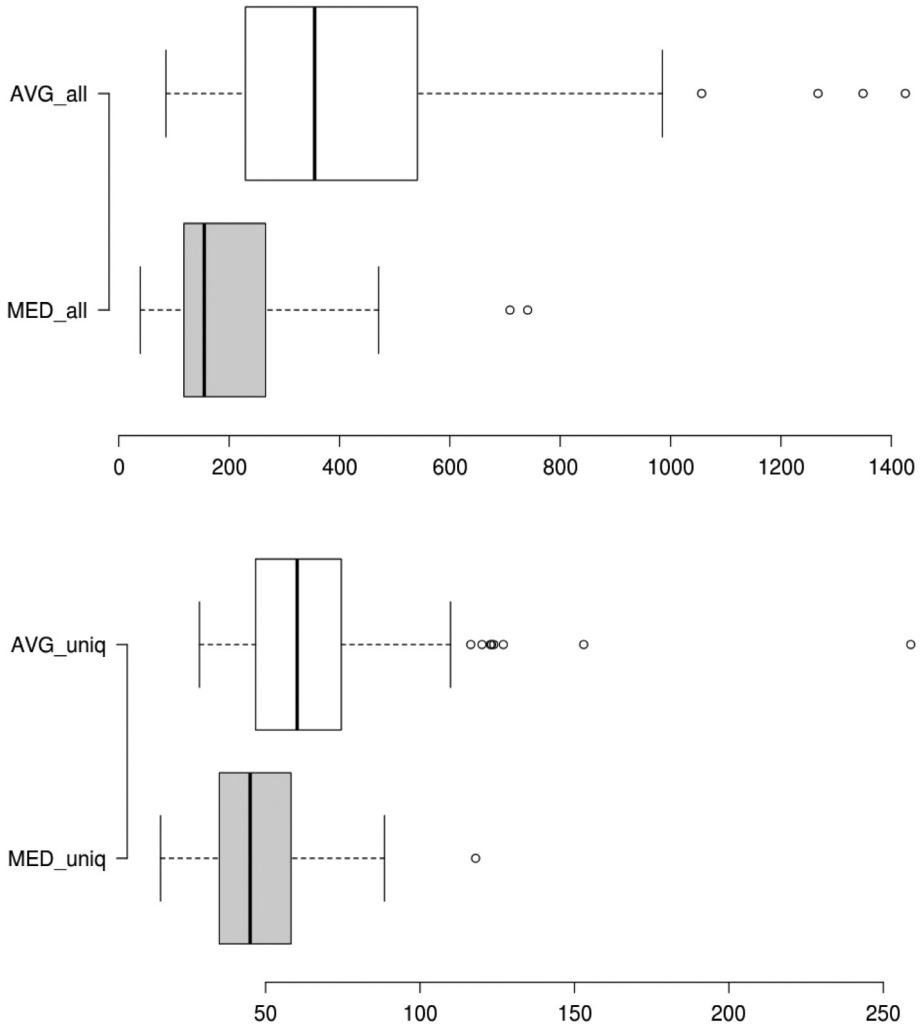

Fig. 5. Average and median values for the *complete* corpora (top) and *unique* corpora (bottom), per project.

average, there are 65 unique terms per Java class, and a median of 45 terms.

We collected the same basic statistics for the subset of test classes only, and for the subset of 'all but test' classes. This is collected in Table 2: as visible in the 'median' and 'average' columns, we could not detect a major difference when considering only test classes, or the overall sample of Java classes. Considering the unique corpus of classes, between 43 and 47 keywords was found to be the median corpus size in any of the subsets (considering or excluding test classes); between 60 and 68 was found to be the average of the corpus of unique keywords. A similar result was found for the complete lexicon corpus of Java classes (with some ~ 180 terms as a median, and some ~ 450 terms as an average).

Project by project Below we analyse the corpora of the Java classes, this time considering each project on its own. We collected the median and average of each project's corpora, and plotted them in Fig. 5.

We observed a larger variability in both medians and averages in single projects, than when considering the overall sample. The *DiskLruCache* project, for example, is composed of only 5 Java classes, but their lexicon complexity is much higher than the rest of the sample, with a median of 118 (unique) terms per class, and 153 terms as the average. Compared to the other systems under study, this is clearly an outlier: these classes should be analysed further and possibly refactored, since their level of complexity is clearly too large. We further analyse this outlier in the Discussion Section 4.5 below.

Clustering by project's size Similarly to what done above, we considered the Java classes from the 'small' projects (e.g., less than 100 classes), separated from the classes of the 'medium' projects

Table 3
Median, average and standard deviation when considering small, medium and large projects.

	Unique corpus			Complete corpus		
	median	avg	std dev	median	avg	std dev
Small projects	46	71	142.26	179	436.5	966.4
Medium projects	44	65	88.32	157	444.69	1331.315
Large projects	46	65	125.5	185	453.27	1544.03

(e.g., between 100 and 1000 classes[14]) and from the 'large' projects (over 1000 classes). This was done to check whether a significant difference could be detected in the groups. The results shown in Table 3 do not point to a distinction between smaller or larger projects: both the median values (46, 44 and 46) and the averages (71, 65 and 65) of the unique corpora (as well as the complete corpora) show similar sets of values.

Based on the Table 3 above, we concluded that there is virtually no relationship between the overall size of the system (as expressed in the number of class) and the size of the *unique* class corpora: this was evaluated using the correlation coefficient between the number of classes of each system, and the median size of the *unique* class corpora, per system. No matter how large or small the Java system is (relatively to our sample), the number of *unique* terms of a Java class remains confined to a range with a small variance.

Hypothesis tests and results The section above produces the following results:

1. our sample shows a limited variability in both the unique and complete corpora of the systems sampled and their classes. For the distribution of the *median* number of unique terms: $\mu_{MED} \pm \sigma_{MED} = 47 \pm 18$.
2. Even considering the split between test and non-test classes, we did not detect a major difference in the size of their corpora.
3. The small variability in the corpus size is irrespective of the size of the system, as evaluated in number of classes.

Given the above observations, we cannot reject the null hypothesis '*the size of a class corpus is quasi-constant among classes, for both complete and unique corpora*'.

3.2. RQ2 – results

Is there a correlation between the unique developers collaborating on a class, and the size of its lexical content?

The second research question that was investigated focuses on the relationship between corpora size and number of developers. The research question is as follows: **Is there a correlation between the unique developers collaborating on a class, and the size of its lexical content?** The null hypothesis states that '*the size of a class corpus does not increase as long as more developers contribute to it*'. We performed two tests: in the first, we considered the complete and the unique corpora of all the classes in the sampled pool. In the second test, we separately considered the corpora of each project.

Overall sample – The evaluation of the Spearman's correlation coefficients (e.g., ρ) produces similar values of when considering the overall sample of classes: 0.2939 (number of developers and size of *unique* corpora), and 0.2931 (number of developers and size of *complete* corpora). These two tests should be considered as statistically significant, since the p-values returned are lower than our selected threshold ($\alpha < 0.05$). In the categories introduced by Marcus and Poshyvanyk (2005), the correlations are considered *low*: from the overall perspective, there is no clear correlation between how many developers work on a Java class, and the size of its corpus.

Project by project We repeated the same correlation study for each project, rather than the overall sample. We evaluated the Spearman's ρ for the size of the corpora and the number of developers specifically working on each project. The value of the correlation coefficient lies in the range $[-1; 1]$, where -1 indicates a strong negative correlation and 1 indicates a strong positive correlation. As mentioned above, we adapted the categorisation for correlation coefficients used in Marcus and Poshyvanyk (2005) ($[0 - 0.1]$ to be *insignificant*, $[0.1 - 0.3]$ *low*, $[0.3 - 0.5]$ *moderate*, $[0.5 - 0.7]$ *large*, $[0.7 - 0.9]$ *very large*, and $[0.9 - 1]$ *almost perfect*) if the rank correlation coefficient proves to be statistically significant at the $\alpha = 0.01$ level.

We collected each project's correlation coefficients in the summary Fig. 6, but only if its Spearman's test was deemed to be statistically significant. In the *insignificant* and *low* categories, we also experienced negative correlations, as highlighted in the hatched patterns.

For the `DiskLruCache` project the correlation between the corpora (complete and unique), and the number of developers modifying the classes is *almost perfect*, with the ρ value at 0.97. In this case, it is clear that a larger number of developers is always connected with a larger corpus of terms in the Java classes that they have worked on.

For 10 other projects (e.g., `dex2jar`, `unirest-java`) there is almost no correlation, with ρ values lower than 0.1 (i.e., *insignificant* correlations). Differently from above, there is virtually no influence of the number of distinct developers over the size of the Java corpora. A similar evaluation can be made for the projects in the *low* category.

In the discussion section below, we will study whether the *application domain* (e.g., 'mobile', 'gaming', 'desktop' etc) of the selected projects plays a role in the type of correlations found in Fig. 6.

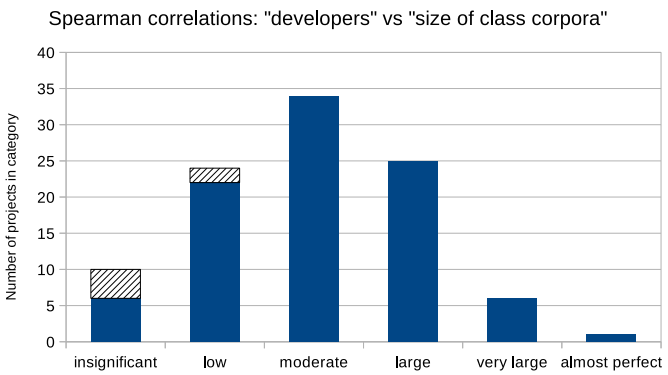

Fig. 6. Spearman's correlations and their categories: size of class corpora and number of developers (per project).

[14] The distribution in the size of classes shows the first quartile at 81 classes, and the third quartile at 834 classes. We rounded up the small projects at 100 classes, the large at 1000 classes.

Table 4
Median and average for the set of unique and complete corpora, when grouped by developer clusters.

Developer clusters	Unique corpus		Complete corpus	
	median	average	median	average
[1]	36	53.11	124	311.72
[2–5]	44	60.54	173	394.10
[6–10]	71	90.44	367	720.97
> 10	121	153.09	881	1629.0

Trends in Clusters Finally, we evaluated the trends between corpora size and number of developers, when considering developer *clusters*. For granularity purposes, we used 4 developer clusters (1 developer; between 2 and 5; between 6 and 10; and over 10). For each of these clusters, we evaluated the average and the median sizes of class corpus, and summarised the trends of their values in Table 4.

The table shows that for the clusters of 1 and [2–5] developers there are some similarities (for the unique corpora the median values are 36 and 44, respectively; average: 53 and 60 respectively). From the clusters [6–10], and > 10, the values are radically higher, and they reflect a stronger influence on the underlying source code (and its corpus) when the number of developers increases.

Hypothesis tests and results The section above produces three interesting results:

1. As an overall sample, we could not detect a direct correlation between the size of the class corpora and the number of distinct developers.
2. At the project level, there is indeed evidence that some projects show the effects of many different developers on the underlying classes. This effect can either be *large* or *very large*, and it is summarised by saying that several developers produce a larger corpus in the Java classes.
3. A further interesting insight was obtained by clustering developers: up to 5 developers do not overly alter the size class corpora, whereas the bracket [6–10] developers (and over) begins to show a clear effect on the size of the class corpora.

Considering the observations above, we cannot reject the null hypothesis '*the size of a class corpus does not increase as long as more developers contribute to the same Java class*' for the overall sample of Java classes.

When considering individual projects, the median value of the correlation distribution shows a strong correlation coefficient. We can reject the null hypothesis, at project level. Section 4.3 below will consider the *application domain* (e.g., security, desktop applications, scientific software, etc) of the projects in our sample as one of the reasons for the observed variability.

3.3. RQ3 – results

The third and final research question that was investigated is the relationship between corpus size and maintenance performed on a class. The research question is as follows: **Is there a correlation between the size of the lexical content of a class, and its maintenance needs?** As "maintenance" we counted the number of changes performed on a Java class, throughout its lifetime (Al Dallal, 2013). The null hypothesis states that '*there is no correlation between the size of class corpora and number of changes of a class*'.

Similarly to RQ2, we also performed two tests: in the first, we considered the corpora of all the classes, while in the second we considered individual projects. We also show the correlation trends when grouping classes in developer *buckets*, for both *complete* and *unique* corpora.

Overall sample – The correlation coefficients that we obtained from the sets 'complete corpora' and 'number of changes' is 0.2549, while the ρ for the 'unique corpora' and 'number of changes' is 0.34879. These results mirror what we obtained for the analysis of RQ2: a *low* (*moderate*) correlation is found between complete (unique) corpora, and the amount of changes performed on a Java class.

The similarity of the results reported for RQ2 and RQ3 implies that there could be a significant correlation between the total number of developers altering a Java class and the resulting amount of changes. This is confirmed evaluating the correlation between "number of developers" and "number of changes" per Java class: the Spearman's test produces a *very strong* correlation coefficient ($\rho = 0.72$) between these two variables.

In the discussion Section 4.4 below we add to this evidence, by considering the experience of developers as a factor for the further maintenance needed: in particular, we will focus on whether less experienced developers are responsible for more added maintenance than the more experienced ones.

Project by project Similarly to RQ2, we focused the correlation study to each project contained in the sample. The Spearman's ρ values were evaluated for the correlation between a class "corpus size" and the "number of changes" that it underwent. We collected each project's correlation coefficient (when the test was statistically significant) in a summary boxplot (Fig. 7).

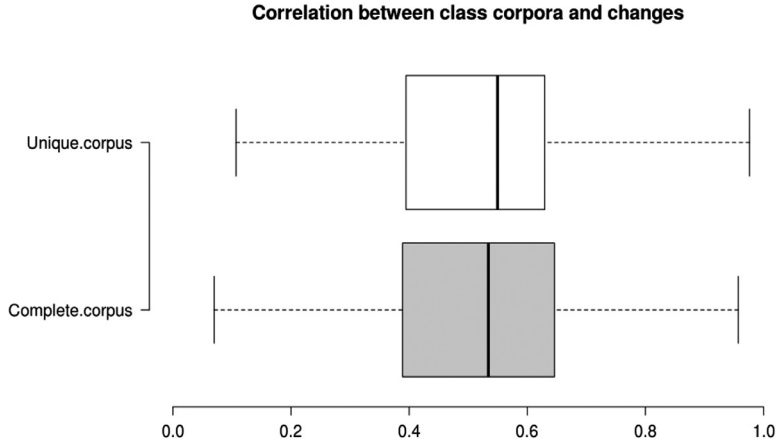

Fig. 7. Correlation coefficients (ρ) between size of class corpora and number of changes (per project).

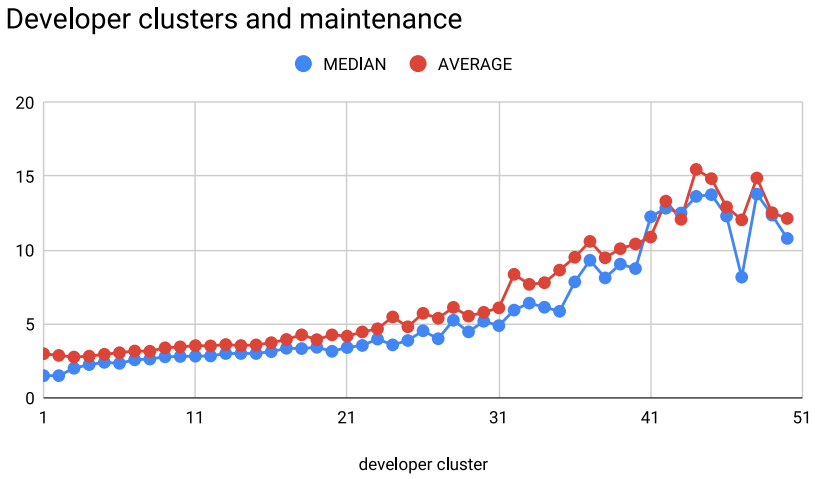

Fig. 8. Growth trends in number of changes, per developer *cluster*.

Single projects have in general a strong correlation between size of the corpus and maintenance needs of a class. As per RQ2, few of the sampled projects show an almost perfect correlation between size of the corpora and number of changes that they underwent. For example, the `DiskLruCache` project displays 0.961 (0.963) as the correlation coefficient between the size of the complete (unique) corpora and the number of changes.

More in general, the size of a system (in number of classes) is a good predictor of the correlation between corpora size and maintenance: for nearly all the projects with *less than* 50 Java classes, the correlation between corpora size and changes is either *very good*, or *almost perfect*. For larger systems, the picture is not as clear: the `pinpoint` project contains over 3700 Java classes, but there is virtually no correlation between complete ($\rho = 0.07$) or unique ($\rho = 0.22$) corpora and the number of changes in Java classes. On the other hand, the `presto` project (over 4300 Java classes) displays *moderate* ($\rho = 0.497$) and *strong* ($\rho = 0.5835$) coefficients for the correlations of *complete* and *unique* corpora with the maintenance changes.

Trends in Clusters Finally, we considered the developer *clusters* as the units of analysis: Fig. 8 shows the trends of the average and median values in the number of changes (the values are normalised by the number of developer in a cluster). As found in the trend analysis of RQ2, when more developers work on Java classes, the number of changes increase. If, on average, Java classes by only one author receive 3 three changes overall, the classes with more than 20 authors get 4 changes. Growing the developer base even larger, classes with 40 authors get more than 10 changes on average.

Hypothesis testing results – Considering the overall sample of Java classes, we cannot reject the null hypothesis: '*there is no correlation between the size of class corpora and number of changes of a class*.

Considering the individual projects, the median correlation value allows to reject the null hypothesis. For smaller projects, the correlation is either *very strong* or *almost perfect*.

4. Discussion and further evidence

In this section we discuss in more detail three aspects emerging from the empirical analyses: (i) the low variance in the size of the unique corpora of Java classes, (ii) the impact of comments and non-code portions on the corpora, and (iii) the effect of multiple developers on the corpora of the affected classes. In order to reduce the bias of the sampling, we also analysed 100 random projects extracted from GitHub, and we repeated the lexical analysis. For each of these discussion points we gathered some further evidence to put them in a wider context, and to expand on their relevance to other researchers and their applicability to practitioners.

4.1. RQ1 – results

Given the findings we summarised analysing RQ1, it becomes important to evaluate the correlation between the size (in both LOCs and SLOCs), and the lexical corpus of the classes. The discussion moves around the question: is the number of keywords correlated to the lines of code?

The distribution of correlation coefficients (between "size in LOCs" and "complete corpora") is displayed in Fig. 9: in terms of LOCs, over 85% of projects show a *nearly perfect* correlation between lines of code and *complete* corpora of their classes. On the other hand, 80% of projects in the sample show a *nearly perfect* correlation between SLOCs and size of the *complete* corpora.

Irrespective of the size measurement used (LOCs or SLOCs), the correlation coefficients between lexical corpora and size are undoubtedly pointing to a strong link between these two entities. The less-than-perfect relationship between SLOCs and *complete* corpora is due to the fact that, per its definition, SLOCs do not consider comments in their count. The terms that appear in the comments of a class do not influence the overall size of the class corpus, in terms of its relationship with size.

4.1.1. Comments and other non-code portions

The results that were collected as part of RQ1 show that the SLOC measure can highlight where the corpus items are used. Fig. 10 below illustrates this relationship graphically: the SLOCs metric (on the right hand side), gets evaluated using only the Code blocks, while the LOCs metric (left hand side) uses all the blocks (Code and Comments).

The sub-corpora sC_1, sC_3 and sC_5 contain the corpus items contained in the source code alone: on the other hand, sC_2 and sC_4 contain the sub-corpora of items contained in the comments. The relationship between the class corpus (i.e., $sC_1 \cup sC_2 \cup sC_3 \cup sC_4 \cup sC_5$) and the SLOCs measure is affected by how SLOCs are evaluated.

When the relationship between SLOCs and lexical corpus becomes negligible, it implies that the amount of key terms contained in the comments has surpassed the key terms contained in the source code. This pattern is linked to a specific code bad smell, the excessive amount of code contained in the class, where comments and documentation are needed to add more explanation to

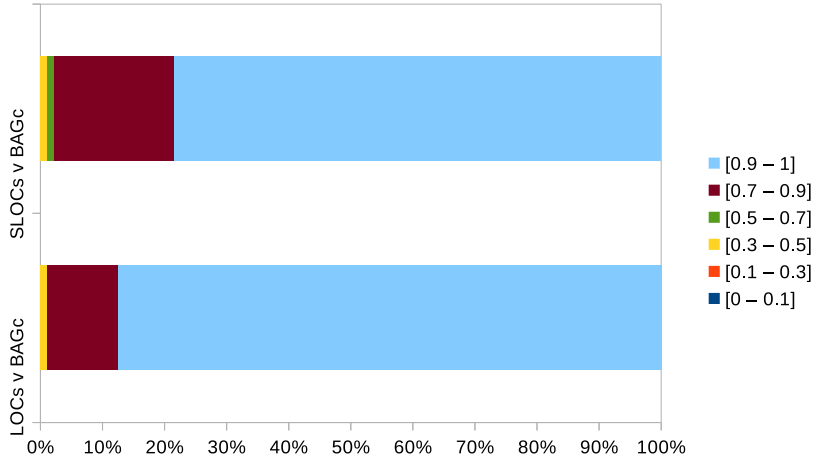

Fig. 9. Correlations between size and *complete* corpora. BAG refers to the corpus of a class (i.e., bag of words).

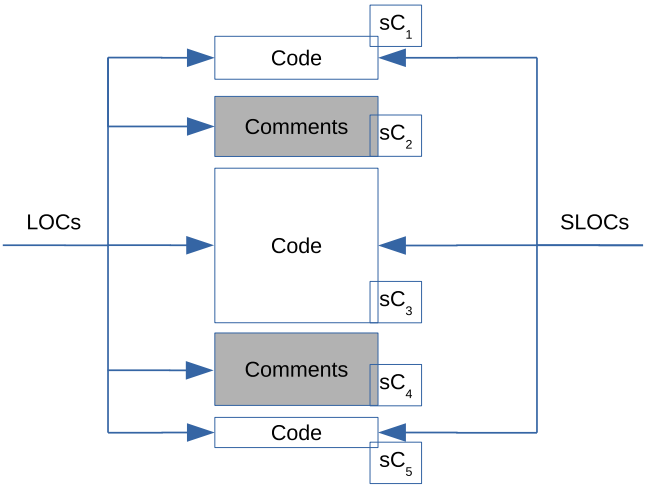

Fig. 10. Evaluation of LOCs and SLOCs and their influence on the class corpus.

the functionality of the code. In this sense, the relationship SLOCs – corpus is a powerful pointer to highlight which classes would benefit from refactoring.

4.1.2. Summary discussion

The analysis that we performed is not exempt from threats to external validity, which we discuss below. Nonetheless, a sample of 83,300 Java classes, from different application domains, can be considered a start to shed some significant light in the research area under RQ1.

We have established that the size of the *unique* corpus of a class can be considered as a source code attribute with low variance. From a previous work, we have also established that the content of the class corpora are fairly distinct from each other (Ajienka and Capiluppi, 2016): the unique terms of one class are rarely overlapping the ones from another class.

Combining the two findings above, we can say that Java classes encapsulate a limited set of terms (or concepts), that are orthogonal to those of other classes. These findings point to further insights: when the class corpus is much larger than any other in a specific software system; and when the same set of terms appear in different classes. In the first case, a split-class refactoring would help in the separation of the concepts of a class that has grown too large, lexically. We discuss a qualitative case below, and how the concepts and lexicon gets more complex when the corpus size of classes gets very large.

In the second case, classes showing similar lexical concepts are most likely linked by an overarching, shared functionality, or even requirement. If the lexical similarity between classes was excessive, a merge-class refactoring would help to coalesce lexically-similar functionalities within a smaller number of classes.

In a prior study (Ajienka et al., 2017) on bridging the gap between conceptual and structural coupling to improve estimated impacts during change impact analysis, we proposed a refactoring approach which extracts chains of textually related functionalities from classes with weak structural coupling links and a weak internal structure (based on semantic cohesion (Marcus and Poshyvanyk, 2005) which is also an important attribute of OO software classes) to form more cohesive classes. Kabaili et al. (2001) state that “some classes have multiple methods that share no variables but perform related functionalities; placing each method in a different class would be against good OO design”.

4.2. RQ2: Corpora and multiple developers

The findings from the size of the class corpora and the cohort of developers have shown that it is not possible to reject the hypothesis of correlation between the two attributes. Larger corpora sizes drive the collaboration of more developers, and vice-versa. Although we could not establish a cause-effect relationship, we noticed that the relationship is not monotonically increasing: with the further increase of developers, the corpora gets stable in size.

As said, the directional relationship between corpora size and developer cohort was not established. This could be related to the type of classes at hand: simpler, smaller classes might have a simpler, smaller corpus. For those classes, the collaboration between multiple developers is not needed, or it will not affect the size of their corpus.

On the other hand, complex classes might require a larger, more complex lexical content: the collaboration between many developers is needed to achieve the implementation of more (lexically) complex classes. Johnson and Foote (1988) have encouraged the design of simpler classes and the splitting of complex ones alike to ease program maintenance and comprehension. On the other hand, comprehension will aid collaboration and contribution in OSS development.

If the relationship is considered the other way, our interpretation of the results changes: the presence of more developers should be considered the *cause* of the increased complexity of the

lexical content of a class. As above, the lexical complexity of a class *plateaus* in the presence of an even large cohort of distributed developers: in those cases, the complexity control acts on top of the ever-growing size of the class, to maintain the corpora size under a threshold.

Our results show that the number of distinct developers that have worked on a class since its creation has a relationship with its lexical content: a higher number of terms per class is observed in the classes with a larger number of developers.

The number of terms does not increase indefinitely. With very large pools of developers, class corpora *cease to increase* in size, in turn becoming smaller, seemingly to ease the collaboration between many developers.

Open Source projects can benefit from these findings, and actively support the collaboration of large pools of developers, by monitoring the lexical content of the Java classes. This will help reduce the required effort to develop or maintain a class.

Prior research shows that on average, there are only about 5–10 distinct terms per method body and 20–50 distinct terms per class. In addition, arbitrary named identifiers make software comprehension and maintenance approaches that rely on lexical content of source code ineffective as identified in a study by Kuhn et al. (2007) wherein arbitrary named identifiers were a threat to the effectiveness of their semantic clustering technique for Object-Oriented software feature identification.

A practical step to improving distributed development and collaboration is an encouraged adoption of the pool of verbs automatically identified by Host and Ostvold (2007) which can serve as a benchmark, dictionary or guide for developers when naming or renaming method and class names in OO software.

4.3. The influence of application domains

In this section, we discuss how the results that we obtained (especially those reported in RQ2) are being influenced by the application domains that the projects belong to.

In order to assign a project to an application domain, we made use of an NLP-based automatic approach to extract the topics from the software systems, and a manual approach to assign the projects to pre-existing categories (Capiluppi and Ajienka, 2019). As the list of categories, we adopted what has been historically used by the SourceForge.net repository to classify the hosted projects: {*1:Communications, 2:Database, 3:Desktop Environment, 4:Education, 5:Formats and Protocols, 6:Games/Entertainment, 7:Internet, 8:Mobile, 9:Multimedia, 10:Office/Business, 11:Other/Nonlisted Topic, 12:Printing, 13:Religion and Philosophy, 14:Scientific/Engineering, 15:Security, 16:Social sciences, 17:Software Development, 18:System, 19:Terminals, 20:Text Editors*}.

In our sample, we found 4 appication domains that are more prominent than others: *Internet* (with 27 projects), *Mobile* (11), *Multimedia* (11) and *Software Development* (34). We report the Spearman's correlation coefficients between number of developers and size of corpora, and we group the results based on the domains. Table 5 contains the number of projects of the different application domains, and we group the correlation coefficients based on the categories described above. As an example, we observed 13 projects in the *Internet* domain that result in a *moderate* correlation between developers and corpora (e.g., research question RQ2).

For most (23 out of 27) of the projects in the *Internet* category the correlations between developers and corpora are between insignificant and moderate; only for 4 projects the correlations are either large or very large. Similarly, for the projects in the Multimedia domain, most of the correlations show low coefficients, and only one project (the *uCrop* project) show a very large correlation, albeit on the lower side of the bracket ($\rho = 0.707$). In a similar fashion, the majority (22 out of 34) of projects in the Software Development domain do not show meaningful correlations between number of developers and size of the corpora.

Table 5
Summary of Spearman's correlation coefficients, grouped in categories: number of developers vs. size of corpora.

Spearman's ρ	Domains			
	Internet	Mobile	Multimedia	Sw Devel
insignificant	1	1	3	3
low	9	0	1	9
moderate	13	3	3	10
large	3	5	3	10
very large	1	2	1	1
almost perfect	0	0	0	1
Total	27	11	11	34

On the other hand, and considering only the *Mobile* projects, most of the correlations are high, and they group as either *large* or *very large*. Projects in this domain show a higher chance to increase their corpora size as long as more developers step in to work on the same Java files.

The two clusters of results can be considered as the starting point of an interesting insight: depending on the domain, Java classes can show a tendency to increase (or not) their corpora, as long as more developers add content to the underlying source code.

4.4. RQ3: corpora and software maintenance

The evaluation of the hypotheses H1 and H2 above pointed to an increase in size of the Java corpora: when more developers have worked on the same Java files, their corpora have deteriorated, according to shared guidelines in software maintenance and evolution.

In this part of the discussion we further analyse what are the repercussions on software maintenance, and whether more developers have an impact on the number of changes that a Java file undergoes, hence its future maintainability. In order to do so, we counted the total number of commits where a Java file was modified, but discounted of the number of developers that worked on each Java files. As an example, the Java file with ID=989,965 was modified by an overall 13 developers, and it received an overall 39 commits in its evolution. Discounting 13 commits (one for each developer), we noted an additional 26 commits that this class received in its maintenance. We repeated this approach for all the Java classes, but avoiding the commits where more than 100 Java files were modified at the same time.[15]

Also, we considered different types of developers, based on the distribution of their Java-based commits in a specific project. Such distribution is typically heavily skewed, with few developers responsible for the majority of the commits (see the bottom part of Fig. 11, that describes the distribution of commits of the *Android-Bootstrap* project).

Based on this distribution, we divided a project's developers in three categories:

- Top Developers (TD) – those developers who committed a total number of Java files larger than the third quartile (Q3) and less or equal the maximum number of Java files;
- Middle Developers (MD) – those developers who committed a number of Java files larger than Q1 but smaller than Q3;
- Bottom Developers (BD) – those developers who committed a number of Java files smaller than Q1;

[15] Those commits are most likely automated commits that change the same portion of code, or even the license, of many source files at the same time, in bulk.

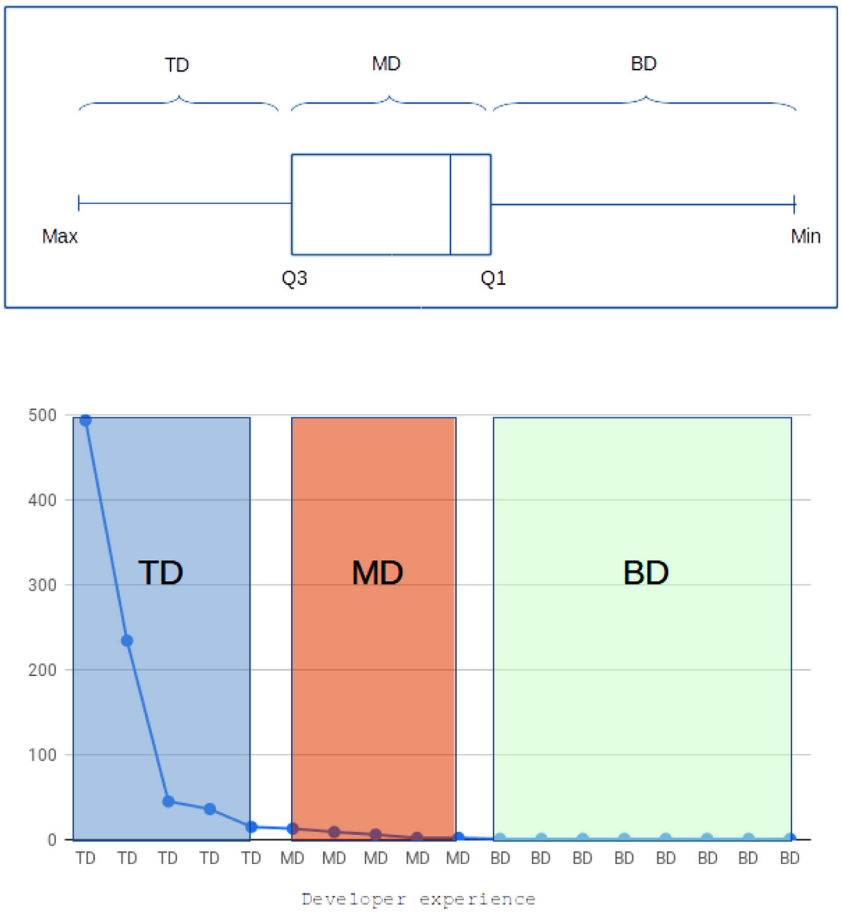

Fig. 11. Extraction of developers experience: boxplot perspective (top) and its evaluation on project ID = 2.

Table 6
Average and median number of additional commits per cluster of developers, and considering experience as a factor.

	ALL JAVA CLASSES			
Dev. clusters	1	2–5	6–10	>10
Further commits (AVG)	0.73	2.73	12.46	75.85
Further commits (MED)	0	1	8	28
	ONLY TOP DEVELOPERS			
Dev. clusters	1	2 to 5	6 to 10	>10
Further commits (AVG)	0.74	2.67	11.56	24.49
Further commits (MED)	0	1	8	17.5
	MD + BD >TD			
Dev. clusters	1	2 to 5	6 to 10	>10
Further commits (AVG)	0.29	2.06	16.82	59.32
Further commits (MED)	0	1	9	37

The results are found in Table 6 below: we separate the scenarios where all the classes are considered; from those where only Top developers are involved; from those where there is a majority of Middle and Bottom developers (e.g., $MD + BD > TD$).

When considering all the Java classes in our sample (section "ALL JAVA CLASSES" in Table 6) we observed that, on average, the classes modified by one developer are those that needed the least further maintenance (less than one additional commit, on average; and no further commits as a median value). When 2–5 developers have worked on a Java class, the additional commits become more than 2 on average, but only one as a median. The additional maintenance becomes much more visible in the "6 to10" developers per class, and extremely high for the classes that were modified by more than 10 developers. In the former scenario, we recorded a median of 8 additional commits; in the latter a median of 28 additional commits.

The same trend is visible when only the Top developers (section "ONLY TOP DEVELOPERS" in Table 6) are involved, but with a difference: the average and median values of additional commits are kept lower than the general case where all developers are considered.

This second finding is confirmed by the analysis of the Middle and Bottom developers (section "$MD + BD > TD$" in Table 6). The development of Java classes by developers with a mixed experience has a visible effect on their future maintenance, requiring a lot more further commits (both in average and median) than when the only Top developers work on the code.

4.5. Corpora size as an indication of class cohesion – qualitative study

In this section we carry out an additional analysis on one specific project in our sample (i.e., `DiskLRuCache`), and we focus on its class corpora, that result in unusually large sizes. For all its classes, we carried out a *manual* inspection on the frequent or re-occurring terms in them. When using the VSM or LSI techniques, the weight (e.g. frequencies) of terms are used to create the 'term by document' matrices. However, given the size of some of the classes, we could not leverage the SVD technique in LSI to extract the topics from the corpora of all classes. Therefore, we reverted to a manual inspection of each class corpus.

The case study that we present below analyses whether very large class corpora contain terms and concepts that belong to too many different domains: in other words, we investigate whether

Table 7
Class corpora size, terms and domains (cohesion). Key - U: number of unique terms per class, C: number of all or complete terms per class.

Class name	Corpus	Terms	Domain
DiskLruCache	U:304 C:2,408	app, backup, buffer, builder, cache, clean, client, close, code, commit, count, create, delete, current, directory, disk, edit, entries, executor, fail, filesystem, exist, index, journal, length, version, write	Multiple domains (commits and version history AND deletion of files AND new entries and creation, backups, etc.)
DiskLruCacheTest	U:231 C:3,160	aa, abc, abort, absent, action, add, allow, backup, bb, buffer, cache, cause, clean, close, commit, confirm, create, delete, directory, disk, edit, equal, evict, except, executor, exist, expect, file, fault, garbage, invalid, journal, line, rebuild, remove, set, sink, size, snapshot, test, trim, version	Multiple domains (commits and versions (and snapshots) AND deletion of files AND new entries and creation of files AND cleaning AND backups AND source code execution AND faults and testing, etc.)
StrictLineReader	U:118 C:456	buffer, capacity, charset, code, count, data, decode, end, except, input, line, read, report	Same domain (text processing)
StrictLineReaderTest	U:72 C:208	ascii, buffer, builder, byte, close, create, end, except, expect, fail, instead, length, line, pad, read, result, span, stream, test	Same domain (text processing)
Util	U:40 C:104	buffer, charset, close, code, content, delete, count, file, read, util, write	Somewhat same domain (file reading and closing OR file reading with some deletion of text before closing)

the size of a class corpus, and the (number of) its reoccurring terms, is an indicator of its cohesion.

Table 7 shows the classes of the `DiskLruCache` project: we list the unique and complete size of each class; the reoccurring terms in the corpora[16]; and an indication of whether the terms indicate a cohesive class which focuses on one domain (e.g., text processing, file reading, networking, etc.), or not.

What we found is that, as the size of a class corpora increases, its cohesion is lost. This is linked with the results in Section 3.2, stating that the corpora size increase as more developers have worked on the classes. We have also identified that larger classes (in terms of their corpora) are in fact in need of further maintenance.

These preliminary results could also be looked at from the perspective of program comprehension. Comments and structure of source code should aid program understanding and reduce maintenance efforts (Khamis et al., 2010). Text mining and conceptual coupling techniques have been used to identify subtle class dependencies not revealed by source code analysis as well as complement source code metrics in software maintenance tasks such as software change impact analysis and change prediction. However, arbitrary terms in the comments of related classes can affect the performance of conceptual similarity tools and techniques (Hotho et al., 2005; Lee et al., 2010). As shown in Table 7, the larger corpora begin to contain arbitrary terms not linked to a domain. On the other hand, in some cases, the class identifiers provide an indication of the terms used within the class which is important for program comprehension and can save time when eyeballing class names in a project or repository. For example, the `StrictLineReader` class identifier can indicate to a developer that the contents or implementation of the class will support text processing. In prior work, we compared the use of only class identifiers to using class corpora for computing the semantic similarity between classes (Ajienka and Capiluppi, 2016) and results from the study revealed that complex corpora provide relatively similar semantic information as class names.

Establishing a new conceptual cohesion metric (i.e., C3) for classes in OO software, Marcus and Poshyvanyk (2005) found a significant correlation between C3 and LCOM5. The C3 of a class is based on the average conceptual similarity of methods implemented in a class. While LCOM5 is based on counting the number of methods referencing attributes in the class. According to Marcus and Poshyvanyk, the idea is similar to the how LSI sums up term frequencies in text documents, and the attribute references are like terms in the methods.

As such, this case study has shown that the C3 metric could be related to the sizes of the corpora of classes and invariably the number of developers that have worked on a class. A further investigation on class corpora size and C3 could provide more insights into the presence or absence of a significant correlation between C3 and other cohesion metrics investigated in Marcus and Poshyvanyk (2005). One can also investigate the effect of the number of developers that have worked on a class and the correlation between the C3 of the class and the other cohesion metrics that rely on source code or information flow.

4.6. Representativeness of the sample

The sample that we extracted from the GitHub repository is a convenience sample, using a stratified approach based on the number of stars received by the projects. The results that we obtained might be biased by the type of systems contained in the sample: also, the results might not be representative to a random sample of Java projects.

In order to assess the extent of the threat to external validity, we analysed a further random sample of Java projects. We considered a curated population of 14,118 Java projects, contained in the research published in Allamanis and Sutton (2013). From that sample we extracted a random sample of 100 Java projects,[17] and repeated the analysis that we performed above.

The random sample contains 16,077 Java files overall, and the projects are sensibly smaller than the original sample presented above: 29 of these projects contain less than or equal to 10 Java files. The violin plot in Fig. 12 shows how the smaller projects skew the distribution, as compared to the original sample.

The original and the new samples are statistically different: a Kolgomorov-Smirnov directional test does not reject the null hypothesis 'the original sample has larger corpora than the random sample'. This is further countered by the Cohen's effect size that we evaluated at $d = 0.1$: the two samples are indeed pooled from different distributions, but their differences are minimal.

[16] We have used a Unix shell script to automatically identify the reoccurring terms in the corpora.

[17] Each project in the population was given an ID, and we extracted 100 random numbers between 1 and 14,118, and avoiding repeated numbers. The list of random Java projects is made available at https://doi.org/10.6084/m9.figshare.11307005.

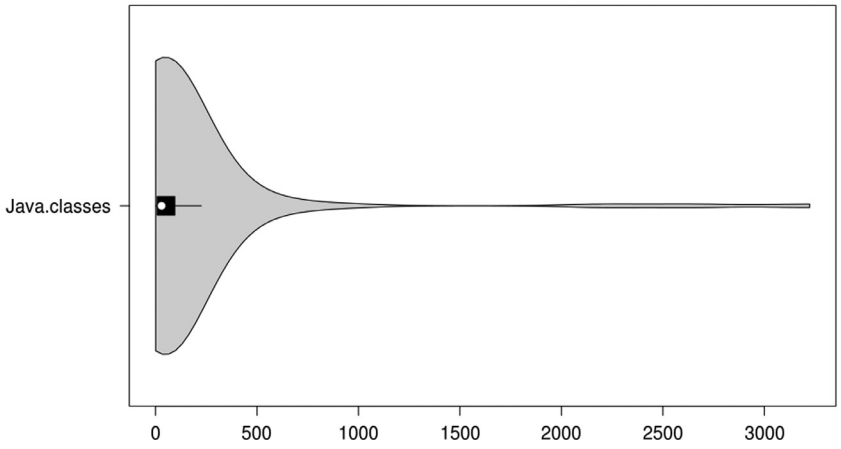

Fig. 12. Number of Java files contained in the projects of the random sample.

Table 8
Median, average and standard deviation (in number of terms) for the unique and complete corpora of Java classes, considering the overall sample.

	Unique corpus			Complete corpus		
	median	avg	std dev	median	avg	std dev
All classes	42	55.78	2,738.88	150	364.09	2.77E+06
Test classes	37	44.93	1,462.42	146	348.40	9.85E+06
All but test classes	44	58.03	2,974.08	151	367.34	1.30E+06

The Java projects with a small number of classes tend to have larger corpora too: as an example, the DAVILA[18] project contains only one large Java class, that contains a total of 1605 terms, and an overall 311 unique terms. As a further example, the JUnitSync[19] project contains 4 Java files, and the median value of the complete corpora is set at 3365 terms, while the median number of unique terms was found to be at 238. These are clear examples that smaller projects tend to have larger classes, that contain a larger lexical complexity, as measured by the terms contained in the classes themselves.

Similarly to the original sample, we provide in Table 8 the distribution statistics for the random sample of Java projects. All the derived values look smaller than the original counterpart: relatively to the unique corpora, the size of the lexicon is still comparable (42 versus 45 as the median value; 56 versus 65 as the average value). The smaller projects, and their Java classes with large corpora, are skewing the values of the variances, that all look much higher than the original sample.

The distribution of unique and comoplete corpora was plotted, per projects, in the two boxplots of Fig. 13.

5. Background and related work

Open-source software, often created voluntarily by developers located in different parts of the world, has been compared to commercial software. This is an unexpected accomplishment as open-source programmers infrequently meet (Yamauchi et al., 2000).

However, challenges still exist with regards to collaboration between dispersed developers. One of such challenges is related to culture and language: since English is traditionally considered the 'lingua franca' of software development, misinterpretations and understandability issues have been flagged (Noll et al., 2010), and practical solutions proposed. The open-source development paradigm has not formally acknowledged the issue in handling international developers, who could be far from native speakers. The mechanisms of such collaboration and what role language plays in facilitating the coordination between developers, have been vastly ignored.

Comments and keywords used by developers for names of classes, methods, or attributes in source code or other artefacts contain important information (Vinz and Etzkorn, 2008; Qusef et al., 2011; Kagdi et al., 2010; Bavota et al., 2014a; 2014b) and account for approximately half of the source code in software (Kagdi et al., 2013). These names often serve as a starting point during program comprehension. Hence, it is essential that these names clearly reflect the concepts that they are supposed to represent, as self-documenting identifiers decrease the time and effort needed to acquire a basic comprehension level for a programming task.

In a related study, Host and Ostvold (2007) emphasise that method identifiers make or break abstractions: while good identifiers communicate the intention of the method, bad ones tend to cause confusion and frustration (Niu et al., 2012). Furthermore, the task of creating identifiers is subject to the sudden ideas and way of thinking of the individual as programmers have little to guide them except their personal experience. Based on this notion, both researchers analysed method implementations taken from a corpus of Java projects, and established the meaning of verbs in method identifiers based on actual use. As a result, they produced an automatically generated, domain-neutral lexicon of verbs, similar to a natural language dictionary, that represents the common usages of many programmers. This lexicon of verbs can serve as a guide for programmers when adding comments and naming or renaming methods and classes in source code.

Seriai et al. (2013) proposed a new approach of identifying functional object-oriented software features. Their approach combines the lexical and structural similarity of classes. Based on their

[18] https://github.com/jabauer/DAVILA.
[19] https://github.com/spring/JUnitSync.

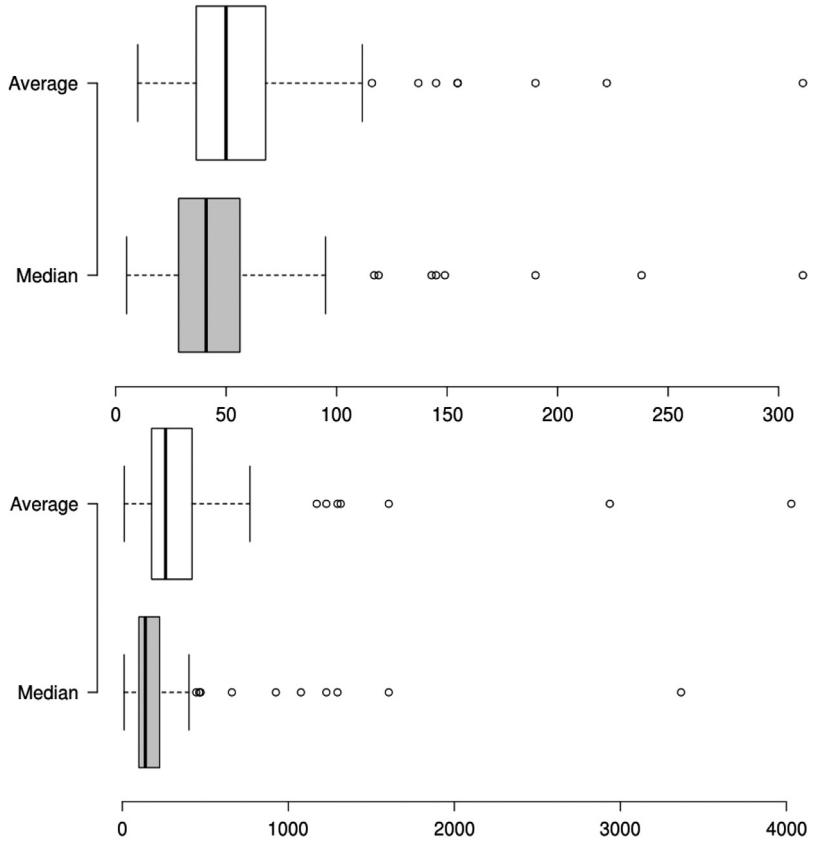

Fig. 13. Average and median values for the unique and complete corpora, clustered by project.

results, they conclude that the proposed method outperforms using lexical similarity only for feature identification.

Aloysius and Arockiam investigated OO software complexity metrics and proposed a Cognitive Weighed Coupling Between Objects (CWCBO) metric which comprises of five coupling types including lexical content coupling (Aloysius and Arockiam, 2012). They defined lexical content coupling as the existence of some or all of the contents of one module in the contents of another. In this paper, we have presented a study on the lexical content of classes in relation to developer collaboration.

Yamauchi et al. (2000) investigated the influence of electronic media (e.g., todo lists and mailing lists) on disperse collaboration among open-source software developers. Their findings suggest that spontaneous work coordinated after the use of electronic media is effective, and rational organizational culture helps achieve agreement among dispersed developers.

Language and culture can influence the way by which people interpret messages (Noll et al., 2010). In the open source software domain, for instance, proficient English speakers might unintentionally intimidate non-English speakers. On the contrary, team members who are not confident with their English language skills may prefer instant messaging or email over telephone or video conferencing, as text-based media provide more time to comprehend and compose a response.

One proposed solution for language barriers is documents authored by non-native speakers of the shared language should be reviewed by a native speaker.

In an attempt to understand what motivates open source software developers, Ye and Kishida (2003) identified that present software engineer education and research emphasise more on program writing. However, from language learning experience, to become good developers, we have to learn to read first. This stresses the importance of the semantics within software artefacts as a major factor for program comprehension which will advance collaboration.

On the naturalness of code, Hindle et al. (2012) emphasise that code written by real people shares characteristics to natural languages including regularities and repetition in utterances which makes it easy for statistical models to be able to predict and complete sentences. This hypothesis applied in software also showed that statistical language models can be used in code suggestion and completion. The authors presented an Eclipse plugin for code suggestion.

Likewise, Ray et al. (2016) argued that real software tends to be similar to natural language: repetitive and predictable and prior researchers have developed code suggestion engines based on this notion. Hence code that tends to be unnatural is possibly faulty or buggy. The authors carried out a study on software projects to assess the naturalness of buggy code vs corresponding bug-fix code. The authors identified that buggy code tends to be more unnatural (lacking entropy or predictability) and becomes less unnatural as fixes are introduced. In addition, source code entropy scores could be useful when predicting defective software components.

Linstead et al. (2007) presented an approach to mine concepts from source code using LDA (Latent Dirichlet Allocation). The authors proposed that the probabilistic relationships between extracted topics and documents also provides a means to measure code similarity.

Similarly, Kuhn et al. (2007) proposed the use of information retrieval to exploit linguistic information in source code, such as identifier names and comments. The proposed technique called Semantic Clustering is based on Latent Semantic Indexing and

clustering to group source artefacts that use similar vocabulary. According to the authors, the proposed approach is language independent as it works at the level of identifier names.

5.1. Semantic dependencies

Semantic coupling captures the degree to which the identifiers and comments from different classes are similar to each other (Poshyvanyk and Marcus, 2006). Thus, it is limited to the underlying meanings of unstructured text in the source code of software entities and how these meanings relate to each other (Gethers and Poshyvanyk, 2010). This relationship can also be quantified, as described in Újházi et al. (2010).

The benefits of the application of semantic technologies to software maintenance have been emphasized in prior research (Rilling et al., 2008). These benefits include software comprehension and traceability recovery (connecting parts of software documentation and source code using information retrieval (IR) techniques). According to Poshyvanyk and Marcus (2006), semantic coupling metrics can be used to "augment existing existing coupling metrics in tasks such as change impact analysis as existing measures do not capture all the ripple effects of changes in software. They also have direct application in reverse engineering tasks like re-modularization". Unlike structural coupling, semantic coupling between class identifiers for example, can be computed independent of programming languages (Újházi et al., 2010).

According to Sharma and Suryanarayana (2016):

- "When *A* and *B* are constructors of classes *X* and *Y* respectively (where *Y* is a sub-class of *X*) such that a change in *A* may impact the object creation of class *Y* through *B*, then *B* is lexically dependent on *A*".
- <ddq>"When *A* and *B* are overloaded methods in a class, a change in method signature of one of the methods may change the method invocation order involving *A* and *B*, then *A* and *B* are lexically dependent on each other"</sdq> (Sharma and Suryanarayana, 2016).

6. Threats to validity

In this section we identify the threats to validity of the results presented in Section 3. Firstly, we cannot generalise our findings on a different sample of OSS projects. Nonetheless, to make our findings more generalisable and representative of OSS projects, we have analyzed the 100 top-ranked projects which are from different domains (e.g., networking, databases, API, etc.) and of different sizes in terms of number of classes, number of lines of code and historical data (number of past revisions). We also analysed a further sample of 100 Java projects, randomly extracted from a population of over 14,000 systems. In this way, we reduced the bias resulting from the original, stratified sampling.

The scope of the studied sample of projects is also limited to OO projects written in Java. This is because of the popularity of the programming language (Burger and Burge, 2016), its cross-platform compatibility (Indig et al., 2018) and the availability of tools to parse and analyse projects written in Java. We encourage investigating projects written in other programming languages, non-object-oriented software projects and commercial software.

Second, we established that the size of class corpora has a *small* variability, claiming a semi-constant pattern for this metric. We used the median value for this result, and the size of the first and third quartiles of the boxplot distribution. Using the average and the standard deviation of the corpora size distribution yields different results: we used the median value given the skewness of such distribution.

Third, we estalished a relationship between corpora size and number of developers: although we could not infer any direction of such a relationship, the interpretation of such findings, and further experimentation, still needs work to produce a better understanding of the dynamics of that relationship.

7. Conclusion and future work

In this paper we focused on the interplay between the lexicon of a Java class (i.e., its dictionary made of keywords) and how that is affected by the collaboration between developers.

We discovered that, overall, Java classes have a similar amount of unique terms. Larger systems, or the presence of test classes, do not appear to modify this underlying trend. This result can serve to direct software development into good practices, and guide and focus refactoring (in particular, splitting classes) once the number of key terms becomes excessive. Our case study confirmed how very large corpora indeed contain terms from different domains and features, that could be split for reducing maintenance, enhancing conceptual cohesion and to increase comprehension.

Secondly we found that, for several projects in the sample, there is a strong correlation between the size of the class corpora, and how many developers worked on them. Further insights were gained from considering specific application domains as the driver of higher correlations.

Thirdly, we showed that larger corpora are linked to more maintenance needs: more terms in a class are associated to more changes to that class. We also showed that there is an exponential growth in number of changes, when dividing Java classes in developer clusters, and that the experience of developers is also linked to the further maintenance of Java classes.

We are considering a further extension of this work: we want to engage in action research with the analysed projects: taking an example of the issues that we uncovered in this paper, we plan to report it to the development team, with a proposed fix. The objective is to understand whether the amendment that we suggest can be useful for the team at large. We also encourage researchers to replicate the study on a different sample of software projects developed in languages other than Java.

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.

CRediT authorship contribution statement

Andrea Capiluppi: Conceptualization, Data curation, Formal analysis, Methodology, Software, Writing - original draft. **Nemitari Ajienka:** Conceptualization, Methodology, Writing - original draft.

References

Ajienka, N., Capiluppi, A., 2016. Semantic coupling between classes: corpora or identifiers? In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, p. 40.

Ajienka, N., Capiluppi, A., Counsell, S., 2017. Managing hidden dependencies in OO software: a study based on open source projects. In: Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on. IEEE, pp. 141–150.

Al Dallal, J., 2013. Object-oriented class maintainability prediction using internal quality attributes. Inf. Softw. Technol. 55 (11), 2028–2048.

Allamanis, M., Sutton, C., 2013. Mining source code repositories at massive scale using language modeling. In: The 10th Working Conference on Mining Software Repositories. IEEE, pp. 207–216.

Aloysius, A., Arockiam, L., 2012. Coupling complexity metric: a cognitive approach. Int. J. Inf. Technol.Comput. Sci. 4 (9), 29–35.

Alshomali, M.A., Hamilton, J.R., Holdsworth, J., Tee, S., 2017. Github: factors influencing project activity levels. In: Proceedings 17th International Conference on Electronic Business, pp. 295–303.

Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., Lucia, A., 2014. Improving software modularization via automated analysis of latent topics and dependencies. ACM Trans. Softw. Eng.Methodol. 23 (1), 4.
Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., De Lucia, A., 2014. Methodbook: recommending move method refactorings via relational topic models. Softw. Eng. IEEE Trans. 40 (7), 671–694.
Bird, C., 2011. Sociotechnical coordination and collaboration in open source software. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, pp. 568–573.
Bird, C., Nagappan, N., 2012. Who? Where? What? Examining distributed development in two large open source projects. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, pp. 237–246.
Borges, H., Valente, M. T., Hora, A., Coelho, J., 2015. On the popularity of GitHub applications: apreliminary note. arXiv:1507.00604.
Borges, H.S., Valente, M.T., 2019. How do developers promote open source projects? Computer 52 (8), 27–33.
Burger, W., Burge, M.J., 2016. Digital Image Processing: an Algorithmic Introduction Using Java. Springer.
Capiluppi, A., Ajienka, N., 2018. National boundaries and semantics of artefacts in open source development. In: Proceedings of the 1st International Workshop on Software Health. IEEE.
Capiluppi, A., Ajienka, N., 2019. The relevance of application domains in empirical findings. In: Proceedings of the 2nd International Workshop on Software Health. IEEE Press, pp. 17–24.
Carmel, E., Agarwal, R., 2001. Tactical approaches for alleviating distance in global software development. IEEE Softw. 18 (2), 22–29.
Carmel, E., Tjia, P., 2005. Offshoring Information Technology: Sourcing and Outsourcing to a Global Workforce. Cambridge University Press.
Crowston, K., Annabi, H., Howison, J., 2003. Defining open source software project success. In: ICIS 2003 Proceedings, p. 28.
Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W., 2015. Modeling readability to improve unit tests. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 107–118.
Devanbu, P., Kudigrama, P., Rubio-González, C., Vasilescu, B., 2017. Timezone and time-of-day variance in github teams: an empirical method and study. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics. ACM, pp. 19–22.
Ebert, C., De Neve, P., 2001. Surviving global software development. IEEE Softw. 18 (2), 62–69.
German, D.M., Manabe, Y., Inoue, K., 2010. A sentence-matching method for automatic license identification of source code files. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, pp. 437–446.
Gethers, M., Poshyvanyk, D., 2010. Using relational topic models to capture coupling among classes in object-oriented software systems. In: Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, pp. 1–10.
Gonzalez-Barahona, J.M., Robles, G., Izquierdo-Cortazar, D., 2016. Determining the geographical distribution of a community by means of a time-zone analysis. In: Proceedings of the 12th International Symposium on Open Colaboration. ACM, p. 3.
Halstead, M.H., et al., 1977. Elements of Software Science, 7. Elsevier, New York.
Herbsleb, J.D., 2007. Global software engineering: the future of socio-technical coordination. In: 2007 Future of Software Engineering. IEEE Computer Society, pp. 188–198.
Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 837–847.
Host, E.W., Ostvold, B.M., 2007. The programmer's lexicon, volume i: the verbs. In: Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on. IEEE, pp. 193–202.
Hotho, A., Nürnberger, A., Paaß, G., 2005. A brief survey of text mining.. In: Ldv Forum, 20. Citeseer, pp. 19–62.
Indig, B., Simonyi, A., Ligeti-Nagy, N., 2018. What's wrong, python?–a visual differ and graph library for NLP in python.
Johnson, R.E., Foote, B., 1988. Designing reusable classes. J. Object-Oriented Program. 1 (2), 22–35.
Kabaili, H., Keller, R.K., Lustman, F., 2001. Cohesion as changeability indicator in object-oriented systems. In: Fifth European Conference on Software Maintenance and Re-engineering. IEEE, pp. 39–46.
Kagdi, H., Gethers, M., Poshyvanyk, D., 2013. Integrating conceptual and logical couplings for change impact analysis in software. Empir. Softw. Eng. 18 (5), 933–969.
Kagdi, H., Gethers, M., Poshyvanyk, D., Collard, M.L., 2010. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In: Reverse Engineering (WCRE), 2010 17th Working Conference on. IEEE, pp. 119–128.
Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2014. The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 92–101.
Khamis, N., Witte, R., Rilling, J., 2010. Automatic quality assessment of source code comments: the JavadocMiner. In: International Conference on Application of Natural Language to Information Systems. Springer, pp. 68–79.
Kogut, B., Metiu, A., 2001. Open-source software development and distributed innovation. Oxf. Rev. Econ. Policy 17 (2), 248–264.
Kuhn, A., Ducasse, S., Gírba, T., 2007. Semantic clustering: identifying topics in source code. Inf. Softw. Technol. 49 (3), 230–243.
Lee, S., Song, J., Kim, Y., 2010. An empirical comparison of four text mining methods. J. Comput. Inf. Syst. 51 (1), 1–10.
Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., Baldi, P., 2007. Mining concepts from code with probabilistic topic models. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 461–464.
Marcus, A., Poshyvanyk, D., 2005. The conceptual cohesion of classes. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). IEEE, pp. 133–142.
Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J., et al., 2004. An information retrieval approach to concept location in source code. In: Reverse Engineering, 2004. Proceedings. 11th Working Conference on. IEEE, pp. 214–223.
Middleton, P., 2001. Lean software development: two case studies. Softw. Qual. J. 9 (4), 241–252.
Niu, N., Savolainen, J., Bhowmik, T., Mahmoud, A., Reddivari, S., 2012. A framework for examining topical locality in object-oriented software. In: Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual. IEEE, pp. 219–224.
Noll, J., Beecham, S., Richardson, I., 2010. Global software development and collaboration: barriers and solutions. ACM Inroads 1 (3), 66–78.
Olson, J.S., Teasley, S., Covi, L., Olson, G., 2002. The (currently) unique advantages of collocated work. In: Distributed Work, pp. 113–135.
Poshyvanyk, D., Marcus, A., 2006. The conceptual coupling metrics for object-oriented systems. In: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on. IEEE, pp. 469–478.
Posnett, D., Hindle, A., Devanbu, P., 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, pp. 73–82.
Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., Binkley, D., 2011. Scotch: test-to-code traceability using slicing and conceptual coupling. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, pp. 63–72.
Rajlich, V., Wilde, N., 2002. The role of concepts in program comprehension. In: Program Comprehension, 2002. Proceedings. 10th International Workshop on. IEEE, pp. 271–278.
Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P., 2016. On the "naturalness" of buggy code. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. IEEE, pp. 428–439.
Richardson, I., Casey, V., Mccaffery, F., Burton, J., Beecham, S., 2012. A process framework for global software engineering teams. Inf. Softw. Technol. 54 (11), 1175–1191.
Rilling, J., Witte, R., Gaševic, D., Pan, J.Z., 2008. Semantic technologies in system maintenance (STSM2008). In: Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. IEEE, pp. 279–282.
Roberts, J.A., Hann, I.-H., Slaughter, S.A., 2006. Understanding the motivations, participation, and performance of open source software developers: a longitudinal study of the apache projects. Manag. Sci. 52 (7), 984–999.
Scholtes, I., Mavrodiev, P., Schweitzer, F., 2016. From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects. Empir. Softw. Eng. 21 (2), 642–683.
Seriai, A.-D., Huchard, M., Urtado, C., Vauttier, S., et al., 2013. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In: Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on. IEEE, pp. 586–593.
Sharma, T., Suryanarayana, G., 2016. Augur: incorporating hidden dependencies and variable granularity in change impact analysis. In: Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. IEEE, pp. 73–78.
Újházi, B., Ferenc, R., Poshyvanyk, D., Gyimóthy, T., 2010. New conceptual coupling and cohesion metrics for object-oriented systems. In: Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on. IEEE, pp. 33–42.
Verner, J.M., Brereton, O.P., Kitchenham, B.A., Turner, M., Niazi, M., 2014. Risks and risk mitigation in global software development: a tertiary study. Inf. Softw. Technol. 56 (1), 54–78.
Vinz, B.L., Etzkorn, L.H., 2008. Improving program comprehension by combining code understanding with comment understanding. Knowl. Based. Syst. 21 (8), 813–825.
Von Krogh, G., Spaeth, S., Lakhani, K.R., 2003. Community, joining, and specialization in open source software innovation: a case study. Res. Policy 32 (7), 1217–1241.
Wettel, R., Lanza, M., 2007. Program comprehension through software habitability. In: Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on. IEEE, pp. 231–240.
Yamauchi, Y., Yokozawa, M., Shinohara, T., Ishida, T., 2000. Collaboration with lean media: how open-source software succeeds. In: Proceedings of the 2000 ACM conference on Computer supported cooperative work. ACM, pp. 329–338.
Ye, Y., Kishida, K., 2003. Toward an understanding of the motivation open source software developers. In: Proceedings of the 25th international conference on software engineering. IEEE Computer Society, pp. 419–429.

Andrea Capiluppi is a Senior Lecturer at Brunel University London (UK). His research interests are Open Source software, software maintenance and evolution, and architectural reuse.

Nemitari Ajienka is a Lecturer at Edge Hill University (UK). His research interests are the logical, semantic and structural couplings of software systems.