# The Effect of Multiple Developers on Structural Attributes: A Study Based on Java Software

Andrea Capiluppi[a], Nemitari Ajienka[b], Steve Counsell[a]

[a] *Department of Computer Science*
*Brunel University London (UK)*
[b] *Department of Computer Science*
*Edge Hill University (UK)*

**Abstract**

**Context**: Long-term software projects employ different software developers who collaborate on shared artifacts. The accumulation of changes pushed by different developers leave traces on the underlying code, that have an effect on its future maintainability, and even reuse.

**Objective**: This study focuses on the how the changes by different developers might have an impact on the code: we investigate whether the work of multiple developers, and their experience, have a visible effect on the structural metrics of the underlying code.

**Method**: We consider nine object-oriented (OO) attributes and we measure them in a GitHub sample containing the top 200 'forked' projects. For each of their classes, we evaluated the number of distinct developers contributing to its source code, and their experience in the project.

**Results**: We show that the presence of multiple developers working on the same class has a visible effect on the chosen OO metrics, and often in the opposite direction to what the guidelines for each attribute suggest. We also show how the *relative experience* of developers in a project plays an important role in the distribution of those metrics, and the future maintenance of the Java classes.

**Conclusions**: Our results show how distributed development has an effect on the structural attributes of a software system and how the experience of developers plays a fundamental role in that effect. We also discover workarounds

and best practices in 4 applied case studies.

*Keywords:* Object oriented, Metrics, Collaborative development, Open source, Software structure

---

## 1. Introduction

Collaborative development, and open source software, have been two major paradigm shifts in software development. Loosely coupled developers coordinate their work via distributed versioning systems, 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. Developers do not need to interact or coordinate their effort: their work, if accepted by the community, leaves traces behind that might have an effect on maintainability for future developers.

The presence of many, different developers in the same project has generally been considered a positive factor [1, 2]. However, there is a dimension that has been studied less often in the evolution and maintenance of OO systems, and this is the effect of multiple developers who worked on the same Java class. The global nature of OSS systems usually allows many developers to work distributedly, and at different times, on the same artefacts. The *branching* feature of most new versioning control systems (e.g., Git) made this feature even more efficient [3].

Very few research papers have analysed in detail the repercussions of having many developers working on the same artifacts, and throughout the evolution of a software system [4, 5, 6, 7].

The idea behind this paper is based on a common scenario: software contributions get stacked on each other over time, and the underlying structure evolves too [8]. What is not clear is how additional developers, with various levels of experience in a specific project, add to that structure, and how that relates to the size or structural complexity of the code.

As a way of an example, and as thrust of this research, let us consider the

`ThriftHiveMetastore.java` file, contained in the Apache *hive* project[1]: 14 different developers have worked so far on its 32 revisions, with commits to the same code repository. Along the changes, the values of structural metrics (for example, those described in the Object-oriented metrics suite in [9]) have also evolved [10, 11]. The inclusion and removal of functionality, modification of condition expressions in control structures, and the insertion and deletion of else-parts of code [12] have resulted in the Coupling Between Objects (CBO) metric of `ThriftHiveMetastore.java` to escalate to a very large value: at its latest revision, the CBO of the class has reached 648[2].

We argue that, if not managed properly, the contributions of multiple developers on the same artifacts could potentially make them more complex than those where only a limited amount of developers make their contributions. From opposite sides of the spectrum, we observed various Java classes that got code contributions from hundreds of developers, and their structural complexity seems unbounded, with attributes that steadily and continuously grow. In other cases, we observed classes that maintained a minimal structural complexity, while still having dozens of new developers joining in the effort.

This paper investigates the effects that multiple developers have had on the structural attributes of Java software, throughout its evolution. We consider a population of over 470,000 Java classes, and we *cluster* them by the number of developers who worked on each during their growth: the one-developer classes are separated from the two-developer classes, three-developer classes and so on. Using the OO metrics on these developer *clusters*, we analysed how each OO metric grows in each of the developer *clusters*. The analysis is exploratory in nature, since no previous studies have attempted to establish a link between OO metrics and number of developers. The two underlying research questions can be articulated as follows:

[1] As available at `https://github.com/apache/hive`

[2] Since the CBO of a class measures the number of other classes coupled to it, the value should be kept low.

1. are OO structural metrics of Java classes invariant to the number of contributions received?
2. is the relative experience of developers in a project a factor for the distribution of the OO metrics?

The remainder of the paper is structured as follows: Section 2 reviews past work on the selected OO metrics: it also formulates a guideline (e.g., 'high', or 'low') for each metric. Section 3 describes the empirical approach that was used to extract the OO metrics as well as the developers, and their experience. Section 4 summarises the results, while Section 5 presents four case studies from our sample, that show how the OO metrics grow, and how contributions change. Section 6 discusses the findings and the threats to validity; Section 7 evaluates the related work, while Section 8 concludes.

## 2. Review of Selected OO Metrics

This section provides a background on the OO software metrics utilised in this paper. For each, we provide a guideline that has been agreed upon by researchers, as a result of past investigations.

In 1994, Chidamber and Kemerer [9] proposed a suite of object-oriented (OO) metrics[3]. It included coupling between objects (CBO)[4], weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion in methods (LCOM). The purpose of these metrics was to provide a theoretical basis for software measures and complexity metrics.

The use of the C&K metrics (and other derived metric suites e.g., Briand's coupling metrics [13]), has become an established field of research [14]. The

[3]Generally referred to as Chidamber and Kemerer Java Metrics (CKJM) or C&K.

[4]Class A is coupled to B if and only if at least one of them acts upon the other, A is said to act upon B if the history of B is affected by A, where history is defined as the chronologically ordered states that a thing traverses in time.

C&K metrics, in particular, were evaluated against the nine complexity metric properties proposed by Weyuker [15] albeit concerns on their efficacy were raised [16, 17].

The C&K metrics have been adopted by researchers in many different scenarios: when predicting software maintainability [18]; studying class dependencies in OO software [19]; evaluating the impact of inheritance types on the metrics [20]; evaluating software cohesion and comprehension [21]; and to validate models to predict failures and defects [22, 23, 24, 25, 26, 27].

### *2.1. WMC (Weighted Methods per Class)*

WMC is a count of the number of methods in a class and is directly linked to Bunges' definition of the complexity of a thing as "the numerosity of its composition" [28]. Chidamber and Kemerer's as well as other researchers outlook on WMC is as follows:

- The larger the number of methods in a class, the greater the potential impact (e.g., lower maintainability) on children, since children will inherit all the methods defined in the class.
- High WMC values could lead to high number of software faults as classes with of a high number of methods are difficult to reuse and maintain [29].

**Guideline**: the WMC attribute should be kept **low**.

### *2.2. DIT (Depth of a class in the Inheritance Tree)*

In OO, the notion of *inheritance* describes a scenario whereby a class (subclass) takes on properties of an ancestor class or base class or superclass. The DIT measures the position of a class in the inheritance hierarchy. In summary:

- The deeper a class is in the hierarchy, the greater the total number of methods it is likely to inherit [9], making its behaviour less predictable [30].
- Khalid *et al.* state that "DIT is directly proportional to complexity" (i.e., an increased DIT will lead to higher maintenance efforts) [31].

**Guideline**: the DIT attribute should be kept **low**.

*2.3. NOC (Number of Children)*

NOC is the count of the number of direct child classes that have inherited properties of (or from) a given parent class [30]. In summary:

- It is related to the scope of properties, and it is a measure of how many sub-classes directly inherit the methods of the parent class [9].
- The higher the number of children, the greater the reuse since inheritance is a form of reuse. However, a higher inheritance means that the class design will become more complex to test [31] due to the influence of the class and number of children.

**Guideline**: the NOC attribute should be in general kept **low**. Higher values could be a direct measure to actively promote reuse within code.

*2.4. CBO (Coupling Between Objects)*

Two classes are coupled if one acts on the other [5] and CBO is the number of other classes coupled to a class. Briand *et al.* [13] described various forms of coupling[6] and defined and compared various mechanisms that constitute software coupling including methods invoking other methods and classes being ancestors of other classes. In summary:

- In order to enhance modularity and promote encapsulation, inter-object class dependencies should be reduced. A large CBO increases the complexity of the system, and it adversely affects other quality factors, such as maintainability, testability and reusability [32].
- A measure of coupling is linked to how complex the testing of various parts of a design are likely to be [19]. The higher the inter-class coupling, the more rigorous the testing needs to be. Excessive coupling between classes is also detrimental to modular design and it limits reuse.

[5] If methods in a class use methods or instance variables defined by another class

[6] Such as message passing coupling (MPC), data abstraction coupling, efferent (Ce) and afferent (Ca) coupling, and information-flow-based coupling (ICP).

**Guideline**: the CBO attribute should be kept **low**.

*2.5. RFC (Response for a Class)*

According to Li and Henry [18] "The response set of a class consists of a count of all local methods and all the methods called by local methods". This number ranges from 0 to N (a positive integer) and is a measure of the potential communication between the class and other classes since it includes methods called from outside the class [9]. As such, if a large number of methods can be invoked in response to a message, the testing and debugging of the class will become more complicated since it requires a greater level of understanding required on the part of the tester.

**Guideline**: the RFC attribute should be kept **low**.

*2.6. LCOM (Lack of Cohesion of the Methods in a class)*

The LCOM metric is based on the notion of the similarity of methods. The degree of similarity of two methods $M_1$ and $M_2$ is the intersection set of instance variables[7] used by both methods for functionality. Based on this notion, the LCOM of a class is the count of method pairs where the intersection set is equal to zero (i.e., a null set) minus the count of method pairs whose similarity is not zero[8]. Researchers outlook on LCOM is as follows:

- Cohesiveness of methods within a class is desirable because it promotes *encapsulation* [9].
- Lack of cohesion implies classes should probably be split into two or more subclasses [33, 34] with cohesive method functionalities.
- Measuring the disparate nature of component methods helps to identify complexity and pitfalls in the design of classes [35, 36].

**Guideline**: the LCOM attribute should be kept **low**.

[7] Member variables declared in a class for which instances of the class own a separate copy.

[8] If the number of similar methods is more than the non-similar methods, then the class is more cohesive.

*2.7. NIM (Count of Instance Methods) and NIV (Count of Instance Variables)*

A method is an operation on an object that is defined as part of the declaration of the class. Every instance of a class has the defined and implemented methods of the class as its properties. The NIM metric has been defined by Lorenz and Kidd [37] as the number of instance methods. These are the methods defined in a class, local to the class [38, 39] and are only accessible through an object of that class.

On the other hand, an *instance variable* stores a unique value in each instance of a class. Destefanis and Counsell [39] defined NIV as the number of instance variables of a class. These are variables defined in a class that are only accessible through an object of that class.

**Guidelines**: similarly to the WMC attribute, the **NIM** and **NIV** attributes should be kept **low**.

*2.8. IFANIN (Count of Base Classes)*

The IFANIN of a class is the number of immediate or direct base classes [39]. In Object-Oriented Programming (OOP), a base class is a class from which other classes are derived or inherit properties from. Therefore, in an inheritance tree the base class(es) of a class will be the class(es) directly above it from which it directly inherits from. In a deep inheritance tree, the same concerns pertaining DIT (as explained in section 2.2) apply to the IFANIN measurement. Differently from the NOC metric (described in section 2.3) which refers to the count of classes derived from a class C, IFANIN refers to the number of classes from which a class D inherits its features from [40].

**Guideline**: the IFANIN attribute should be kept **low**.

## 3. Empirical Approach

The study presented here is based on the collection of Java classes, their OO metrics 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 in this section, together with a working example.

The dump of the database used for extracting the results is made available under `https://figshare.com/projects/OO_metrics_vs_Developers/60404`. A replication package is made available at `https://github.com/acapiluppi/oometrics_developers.git`.

### *3.1. Hypotheses*

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

$H_{0,1}$ the OO metrics correlate between them, independently of the number of developers modifying the classes.

**Test**: this hypothesis will be tested by means of a Spearman's $\rho$ test.

$H_{0,2}$ the value of individual OO attributes do not change, as long as more developers contribute to the same Java class.

**Test**: this hypothesis is tested by the growth trends of the OO attributes, depending on the number of developers. The OO attributes of the classes developed by, e.g., one developer will be tested against the attributes of the classes developed by two developers, three developers and so on.

$H_{0,3}$ the value of individual OO attributes do not change, as long as developers *with different experience* contribute to the same Java class.

**Test**: similar to the hypothesis above, this hypothesis is tested using by the growth trends of the OO attributes, but using the relative experience of a developer in a project as a factor.

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. We adapt the categorisation for correlation coefficients used in [41] ($[0 - 0.1]$ 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.

The correlation between any two vectors is assessed using the Spearman's rank correlation coefficient [42]. Spearman's rank correlation is a non-parametric test and is chosen because neither the OO metrics, nor the number of developers per class, has a normal distribution overall, and in each project. We tested each OO metric for normality, using the Kolgomorov-Smirnov test: we could reject the probability of these distributions to be associated to a normal distribution with p-values lower than our threshold ($\alpha = 0.05$).

Various correlation coefficients have been considered including Pearson, Kendall and Spearman. Nevertheless, for Pearson's to be valid the data has to follow a normal distribution [42, 43] (the mean, median and mode have to be the same) while Kendall's tau is adopted in scenarios with small sample sizes and where there are multiple values with the same score [44] and interpreted based on the probability of concordant and discordant observations. In addition, p-values derived from Kendall's tau are more accurate with smaller sample sizes.

### *3.2. Dataset used*

In this study, we have investigated the link between the structural attributes and collaboration in OO software. Leveraging the GitHub repository, we collected the project IDs of the 200 most forked Java projects hosted on GitHub as case studies. As such, our data set does not represent a random sample, but a stratified sample based on one attribute (i.e., forking) that is related to successful development. Other GitHub attributes might be more related to the successful usage of individual projects (e.g., the number of *stars* that it received from other users); the 'number of forks' attribute is an indirect measure of parallel development, since it shows how many further developers decided to contribute to the project.

As a result of the data extraction, we collected 474,197 Java classes, contained in 293,047 Java files. The SQL dump of this data is available at `https://figshare.com/projects/OO_metrics_vs_Developers/60404`.

The repository of each project was downloaded and stored, with its metadata (i.e, the list of revisions for each class, and for the whole project, the developer

IDs, as well as the date and time of each change), using the CVSAnalY set of tools[9,10]. These revisions do not contain files without the .java extension[11].

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.

The largest number of revisions was found in the *elasticsearch* project, with over 89,000 revisions, while the median of the number of revisions *per* project is 2,000. The project with the larger number of classes is a similar value is found for the median number of `.java` classes *per* project.

### *3.3. Size: number of classes and SLOCs*

The 200 selected systems are all mostly written in Java, but the number of classes contained in each system varies: a small number of outliers shows a number of classes to be larger than 2,000; most systems were considerably smaller. The average number of classes in that set was 473, while the median of the set was 166 classes.

A correlation was computed between the number of classes and the number of revisions: a Pearson correlation test ($\rho$) was performed between the set of values representing the number of revisions, and the set of values with the number of classes. We observed that the number of classes and the number

[9] `http://metricsgrimoire.github.io/CVSAnalY/`

[10] Installation steps can be found at: `https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/howtoinstallandruncvsanaly2inubuntu1110`

[11] All the raw data, contained in SQL tables, is hosted at `https://figshare.com/articles/MySQL_dump_of_analysed_projects/9988553`.

of revisions are strongly correlated ($rho = 0.88$): larger systems (in number of Java classes) are more likely to undergo a larger number of revisions, i.e. their historical maintenance work has been much larger.

The size of each class was also measured counting the source lines of code (SLOCs), per Java file, using the *cloc* tool[12], that aggregate the lines of code and separates them from comments and blank spaces.

### *3.4. Extraction of OO attributes*

The OO attributes were extracted using the Scitools Understand tool[13], that extracts each C&K attribute, together with the NIM and NIV attributes too. Abstract classes, interfaces and inner classes were also considered in the data extraction.

The pair ("$project\_ID$", "$full\_path\_of\_Java\_class$") was used as the primary key of the SQL table containing the OO attributes. This was later matched with the same pair, as extracted from the table containing the information of how many developers worked on each class, per project. The scripts to reproduce this step are available in the GitHub project at `https://github.com/acapiluppi/oometrics_developers`.

### *3.5. Extraction of developer metadata*

All the projects in the case study presented below are taken from the GitHub online repository. Several developers are currently working on each of those projects in parallel: in particular, the mechanism of the project *forking* facilitates the parallel development, and collaboration on different classes. For the purpose of this paper, we have counted the number of *distinct* developers who have modified at some point any parts of a Java class.

The Git mechanics allow to log the metadata of individuals as either *committers* or *authors*: in the former case, these are the individuals who actually committed the code in the code-base, but they might have not written it in the

[12] `http://cloc.sourceforge.net/`
[13] `https://scitools.com/`

first place. In the latter case, individuals are acknowledged and mentioned as authors, whilst not being committers to the code-base: this is the typical case where a branch was successfully *merged* in the main trunk. Our definition of *developers* is based on the data gathered on the *authors* of each system.

*3.5.1. 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. The Perl script that achieve these steps are shared in the GitHub project `https://github.com/acapiluppi/oometrics_developers`, for inspection and potential further contributions by other interested researchers.

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 both the *Name* and *Surname* fields of the former, and looked for the same pattern in the latter. This means that each surname in the form 1), e.g. 'Smith', was lower-cased, and looked up via a regular expression search on all the monikers of form 2). The same process was applied for the names of form 1). A sample of these cases was manually verified. 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, where 'Travis' retrieves the 'travisc' moniker via a regular expression.

The script that performs the reconciliation of names from a project, starting from the metadata stored by CVSAnalY, is available inside the replication package at `https://github.com/acapiluppi/oometrics_developers`.

Table 1: Reconciliation of duplicate IDs in the developers metadata

| **project** | **Dev name** | **Dev ID** | **Reconciled dev ID** |
|---|---|---|---|
| robolectric | petrcermak | 11136 | 11015 |
| robolectric | cermak | 11015 | 11015 |
| robolectric | Travis Collins | 10894 | 10894 |
| robolectric | travisc | 11097 | 10894 |

Figure 1 shows the average and median number of authors per Java class, when considering each of the analysed project. The graph shows that a large number of projects (99 out of 200) have one single developer as the middle of the developers' distribution (i.e., median = 1). Another 69 out of 200 projects have a duo of developers as the median.

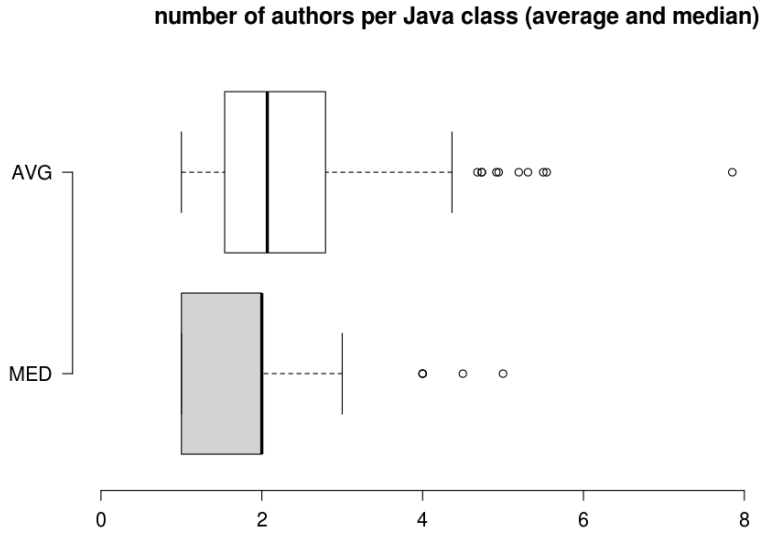



Figure 1: Average and median number of developers per Java class, and per project

We investigated whether larger projects (in terms of overall number of Java

classes) could be connected to a lower average (or median) number of developers per class: the correlation found was very weak for both average and median (0.03 and -0.042, respectively). We concluded that the size of the software systems in our sample is not a predictor of how many developers on average work on their classes.

The tables with the base and reconciled IDs, together with the reconciling script, are made available in the shared repository under `https://figshare.com/projects/OO_metrics_vs_Developers/60404`, for inspection and feedback.

### *3.5.2. Developer clusters*

Figure 2 illustrates the data extraction for two example projects, M and N: in project M, class A has been modified by 3 developers, while B and C by one developer only. In project N, class D has also been modified by only one developer, E and G by two developers, and F by three developers.

Classes B, C and D store their OO metrics (shown in the green colored squares beside each class) in the same *cluster*; the same applies for classes E and G whose corpora are stored in the two-developer *cluster*. Finally, the OO metrics of classes A and F are stored in the three-developer *cluster*.

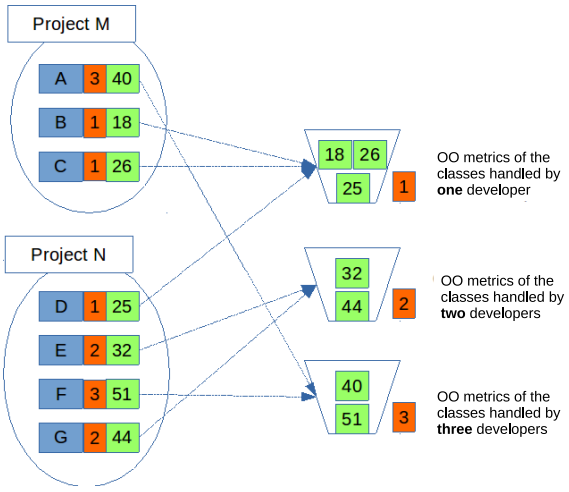



Figure 2: Assignment of class corpora to developer *clusters*

From the projects analysed, we observe that the size of these *clusters* is heavily biased: out of an overall 474,197 classes, there are 127,314 classes that have been modified by only one developer; 78,680 are modified by 2 developers, and 54,837 modified by 3 developers. For the sake of coarseness, in the empirical analysis we used the following clusters:

1. Java classes worked on by *one developer* only;
2. Java classes worked on by *2 to 5 developers*;
3. Java classes worked on by *6 to 10 developers*;
4. Java classes worked on by *more than 10 developers*.

The one-developer cluster identifies classes that are either very simple (thus not needing further contributions), or very complex (such that other developers do not feel like contributing [5]. The cluster '2 to 5' developers helps in isolating the work that is traditionally considered the remit of *small* teams [45]. We use these categories to separate medium-sized teams (between 6 and 10 developers) from larger teams (over 10) [46]. Similar categorisation has been adopted in prior research [47, 48].

### *3.6. Deriving developers experience*

Apart from dealing with duplicate authors, and devising a method to deal with them (see section 3.5.1 above), we also designed an approach to evaluate the relative experience of developers in a specific project. This way, we can tune our previous results in a more specific scenario, specifically dealing with how (project-specific) experienced and less-experienced developers collaborate, and whether experience plays a role. It is important to notice that we did not measure the overall (or personal) experience of any developer, but just their experience relatively to the project under investigation.

We describe our approach in the steps below: it is based on a project-by-project basis.

1. First, we considered all the commits that affected Java source files (i.e., where the files committed had a ".java" extension) in every project of our sample;

2. we excluded those commits that modified more than 100 Java files in the same commit[14];
3. using the remaining commits, we derived, per developer, the sum of different (*e.g.,distinct*) Java files that they worked on[15];
4. using this sum, for all developers in a project, we created a distribution, and evaluated its minimum, maximum, together with the first and third quartiles (see the boxplot in Figure 3 (top));
5. 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;

The definitions of TD, MD and BD are suggested by a recurring type of distribution of developers' effort, and its skewness: this is shown in an example project (e.g., project ID = 2) in the graph of Figure 3 (bottom). Few developers work on the large majority of Java files, and that clearly separates them from the other two types of developers (the trend represents the distribution of developers' experience for project ID = 2).

Considering the sample of analysed projects, we found that the proportion

[14]There are 6,143 commits in our database that, alone, modify or amend over 100 Java files: in the majority of those commits, the message by the developer mentions "moving" or "move", hence the commit can be considered as non-maintenance related. Commits affecting over 1,000 Java files are typically the very first commit onto the GitHub platform, or license updates.

[15]The *file_copies* database table keeps track of files that have been 'moved' or 'copied', so that we can follow the same file with different IDs.

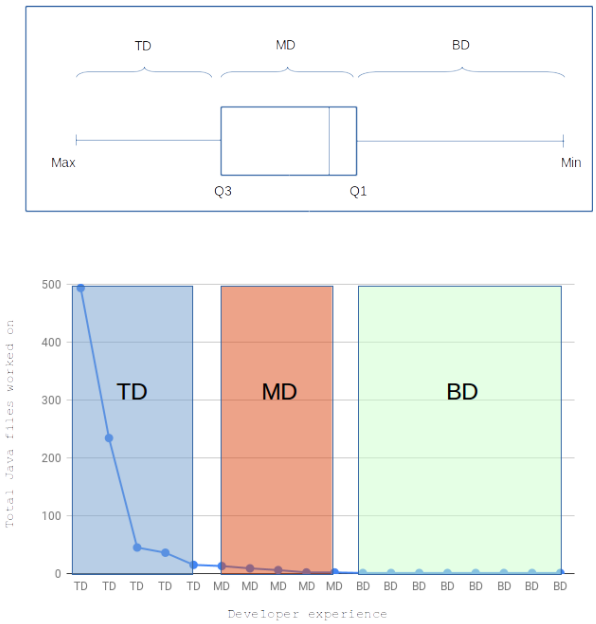



Figure 3: Extraction of developers experience: boxplot perspective (top) and its evaluation on project ID = 2

of top developers (TD) has a low variability (see the TD boxplot of Figure 4 for the vast majority of projects: around 1 in 4 (25%) of developers are in the top spectrum. Coupled with how the TD term is evaluated, it is possible to summarise that some 25% of every development team in our sample is responsible for 75% (and over) of the Java classes in a system.

The remaining 75% of a development team is evenly distributed between the MD and BD types of developers: the middle tier of developers spreads between 20% and 55% of a project's team, with the median at 41% of developers (as in the MD boxplot of Figure 4; whereas the BD tier of developers has a lower median (33%).

In order to study the third research hypothesis $H_{3,0}$, we firstly considered the scenario where only top developers worked on the Java code: we created the buckets of files touched by 1 top developer, 2 top developers etc; and we analysed the trends of the OO metrics described above.

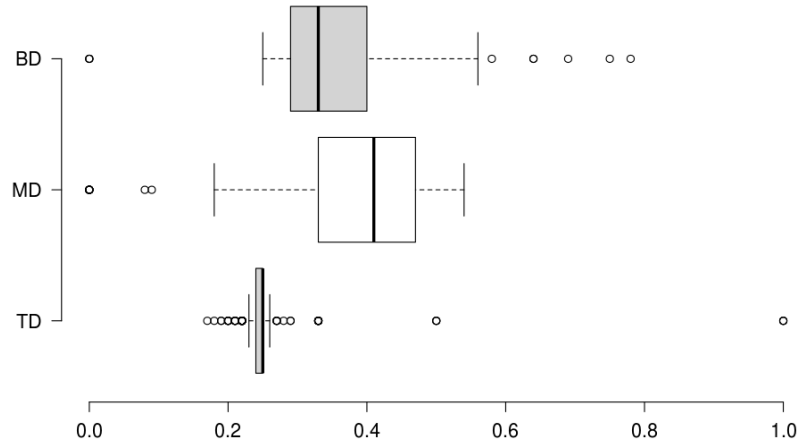



Figure 4: Rates of Top (TD), Middle (MD) and Bottom (BD) developers, per project

Secondly, we considered two further scenarios where Java files have been worked on by a *team* of Top, Middle and Bottom developers: one with a majority of Top developers (e.g., $TD > (MD + BD)$); and one with a majority of either Middle or Bottom developers (e.g., $(MD + BD) > TD$). Also in those cases we produced the buckets of 1 developer, 2 developers and so on.

## 4. Results

In this section we present the results that we obtained running the first two tests. We group the findings by the hypotheses that were presented in the sections above: in section 4.2 we investigate whether the C&K metrics of the Java classes show some significant correlation between each other, considering all the classes in our sample, or the developer clusters (section 4.3). This analysis is not purely a correlation study: it will show how developer clusters might be useful to put past research into a new perspective (as discussed in section 6.1).

Section 4.4 presents the results of the second research hypothesis (H2), and it shows the trends that we observed while plotting the values of each C&K metric against the number of developers. Section 4.5 deals with the third hypothesis (H3) and it evaluates the effects of the experience of developers (relative to the project that they contributed to) on the distribution of OO metrics.

*4.1. Relationship between SLOCs, OO Metrics and Contribution Teams – $H_{0,1}$*

Each Java class produces a set of 9 measurements related to the selected OO metrics. We evaluated the Spearman's correlation between each metric and the size of the class in SLOCs, to determine if there is indeed a correlation between OO attributes and lines of code. We could only consider the Java files containing one class (some 215k Java files, out of a total of 270k in the sample): in the case of multiple classes within the same Java file, each class would produce a different set of OO measurements, but we could collect only the size in SLOCs of the overall file.

We group the correlation coefficients into the intervals defined by [41]. We do accept that other intervals for labelling the strength of correlation are perfectly reasonable (the process is largely subjective), but use the previous definitions simply to remain consistent with that work and to also allow comparisons with the same work to be made. We note that none of the correlation tests was deemed to be non-statistically significant.

At the project level, we obtained a distribution of correlation coefficients, one per OO metric. As an example, Table 2 summarises the *mockito* project[16].

Table 2: Spearman's correlations (and their relative correlation intervals) between each OO attribute and SLOCs (*mockito* project)

| | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| $\rho$ | 0.17 | 0.35 | 0.21 | -0.27 | 0.35 | 0.49 | 0.37 | 0.31 | 0.23 |
| band | l | M | l | (l) | M | M | M | M | l |

The correlations that we observe for the *mockito* example project are consistently either of low or medium strength, the hierarchical metrics (e.g., NOC and DIT) showing a low correlation with the lines of code of the affected classes.

When considering all the classes in the sample we obtained a similar distribution of correlations: overall, the IFANIN, CBO, NIM and LCOM have all low (or insignificant) correlations with the SLOCs, while as seen for the *mock-*

[16] `https://github.com/mockito/mockito`

*ito* example project, the NOC, NIV, WMC, DIT and RFC lie in the moderate correlation band.

Table 3: Spearman's correlations between OO metrics and SLOCs (overall sample)

| | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| $\rho$ | 0.032 | 0.317 | 0.116 | -0.15 | 0.368 | 0.473 | 0.428 | 0.304 | 0.110 |
| band | i | M | l | (l) | M | M | M | M | l |

We conclude that:

none of the OO structural metrics is strongly correlated with the size of the classes, when evaluated in source lines of code (SLOCs)

#### *4.1.1. Relationship between SLOCs and contribution teams*

Finally, we also measured whether the lines of source code in a Java file have any relationship with the number of different contributors to Java file. For the overall sample, we obtained a Spearman's $\rho$ of 0.231 between the two attributes (with statistical significance granted to the test). We concluded that:

there is no correlation between the size of classes in SLOCs and the size of their contribution teams

### *4.2. Correlation between C&K metrics*

In this section, we consider the overall sample of Java classes: each OO metric was extracted for all classes, and correlation coefficients evaluated for each pair of OO metrics. Table 4 shows the results of the correlation (Spearman's test) between the metrics extracted: all tests were statistically significant.

- *Almost perfect* (0.9 - 1]: no relationship between C&K metrics was observed in this category.
- *Very large* (0.7 - 0.9]: there is only one relationship whose correlation shows a very large coefficient, and that is the pair (NIM v WMC).

- *Large* (0.5 - 0.7]: several pairs of attributes show a large correlation coefficient, as provided by the Spearman's $\rho$. The majority of these pairs are composed of intra-class OO attributes (e.g. RFC v NIM, RFC v WMC, WMC v LCOM and NIM v NIM); the pair RFC v DIT on the other hand, also includes inter-classes OO attributes (e.g. DIT).

- *Moderate* (0.3 - 0.5]: as above, many pairs of OO attributes show correlation coefficients in the moderate category. Most of these pairs include either CBO, LCOM or NIV.

- *Low* (0.1 - 0.3]: one third of the pairs of OO attributes (12 out of 36) shows a correlation coefficient in the low range. The IFANIN in particular, is an attribute that correlates quite weakly to the other OO attributes (apart with NIV). Similarly, the NOC attribute shows weak or insignificant links with any of the other metrics.

- *Insignificant* (0 - 0.1]: one fourth of the pairs of OO attributes (9 out of 36) show an insignificant correlation coefficient. NOC and DIT are the two attributes that show the lowest correlation coefficients with the other OO metrics. The only exception is the DIT v RFC relationship that manifests a large (L) correlation between the two attributes.

From the correlations between OO metrics, we observed that:

most of the OO metrics do not correlate with each other, apart from those that, directly or indirectly, measure the number of OO methods

### *4.3. Spearman's Correlation – Developer Clusters*

Section 4.2 has shown the correlations between OO attributes for the *overall* sample of Java classes. This section analyses the relationship between OO attributes when there is *more than one* developer developing the code of a Java class.

We grouped developers into the following further clusters: 2 to 5 developers, 6 to 10 developers, and more than 10 developers. Table 5 summarises the

Table 4: Correlation types between C&K metrics, when all the classes are considered. Highlighted the "very large" and "large" correlations

| | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT |
|---|---|---|---|---|---|---|---|---|
| CBO | 0.080 | 1 | | | | | | |
| NOC | -0.180 | -0.049 | 1 | | | | | |
| NIM | 0.197 | 0.316 | 0.119 | 1 | | | | |
| NIV | 0.223 | 0.161 | -0.016 | 0.564 | 1 | | | |
| WMC | 0.184 | 0.378 | 0.093 | 0.915 | 0.521 | 1 | | |
| RFC | 0.013 | 0.298 | 0.038 | 0.613 | 0.274 | 0.628 | 1 | |
| DIT | -0.127 | 0.073 | -0.140 | 0.081 | -0.093 | 0.007 | 0.535 | 1 |
| LCOM | 0.249 | 0.253 | -0.043 | 0.510 | 0.596 | 0.591 | 0.367 | -0.011 |

correlation coefficients in the proposed bands (*insignificant*, *low*, *moderate*, etc), and how they change when more developers are working on the same Java class. In the table, we highlight in grey the relationships that change at least once in any of the developer clusters. We ordered the table by the correlation bands (insignificant, low, moderate, etc).

When the developers increase, we observed that several correlations change (once or more) their correlation bands, as compared to the overall sample. As an example, the IFANIN v NIM correlation coefficient increases to a *Moderate* (up from *low*) correlation coefficient level when the number of developers working on the classes is larger than 10.

We also observed that certain OO attributes are more prone to change their correlation bands: IFANIN, LCOM and RFC (WMC and DIT to a lesser extent) are the attributes that show the largest variability in the correlation with another attribute. On the other hand, CBO not only shows a very low correlation with any of the other attributes, but its correlation levels do not change as long as more or less developers develop the Java classes. Finally, the boundary values in developer clusters (e.g., only one developer, and more than 10 developers) drive most of the variability of the correlation bands: as an example, the RFC v DIT correlation drops to a *moderate* level for the classes developed by one developer, while it stays in a *large* band for all the other developer clusters.

Table 5: Bands of correlation coefficients in four developer clusters (only 1 developer; 2 to 5 developers; 6 to 10 developers; and more than 10 developers), as compared to the overall class sample

| | | Developer clusters | | | |
|---|---|---|---|---|---|
| OO attribute pairs | All classes | 1 | 2 to 5 | 6 to 10 | Over 10 |
| NOC v WMC | i | i | i | l | l |
| IFANIN v RFC | i | l | i | -l | -l |
| NOC v RFC | i | i | i | i | l |
| CBO v DIT | i | -l | l | i | i |
| NIM v DIT | i | l | i | i | i |
| WMC v DIT | i | i | -l | -l | i |
| IFANIN v NIV | l | M | l | l | l |
| CBO v NIV | l | i | l | l | M |
| CBO v RFC | l | l | M | M | L |
| NIV v RFC | l | M | l | l | M |
| IFANIN v LCOM | l | M | l | l | l |
| CBO v LCOM | l | l | l | M | M |
| CBO v NOC | -l | -l | -l | i | i |
| NOC v NIV | -l | -l | -l | i | l |
| NIV v DIT | -l | -l | -M | -M | -l |
| NOC v LCOM | -l | -M | -l | i | l |
| DIT v LCOM | -l | i | -l | -l | -l |
| CBO v WMC | M | l | M | M | L |
| CBO v NIM | M | l | M | M | L |
| RFC v LCOM | M | M | M | M | L |
| IFANIN v NOC | -M | -M | -M | -M | -l |
| IFANIN v DIT | -M | -l | -l | -M | -M |
| NOC v DIT | -M | -M | -M | -l | -l |
| NIV v WMC | L | L | M | L | L |
| WMC v RFC | L | XL | L | L | XL |
| RFC v DIT | L | L | L | L | M |
| NIM v LCOM | L | M | L | L | L |
| NIV v LCOM | L | M | L | L | L |
| NIM v WMC | AP | XL | AP | AP | AP |

We concluded that:

most of the correlations between OO metrics are affected by the number of developers who contributed to the classes

### *4.4. OO Metrics and Developers – $H_{0,2}$*

In this section we report on the analysis that we carried out regarding the relationship between single OO metrics and number of developers. We analysed the subset of OO metrics as clustered by number of developers, and extracted the average, median and variance of the subset, per OO metric, and per cluster.

Table 6 displays the trends of each OO metric, when considering the clusters (one developer, two developers and so on) of code contributions to Java classes. For example, all the CBO measurements of the classes modified by at most one developer were pooled together and averaged to 4,670.

Table 6: Growth of the OO metrics in the developer clusters (average values)

| Dev's | CBO | DIT | IFANIN | LCOM | NIM | NIV | NOC | RFC | WMC |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4.670 | 1.759 | 1.371 | 29.750 | 5.360 | 1.404 | 0.627 | 19.840 | 5.919 |
| 2 | 4.866 | 1.783 | 1.275 | 26.708 | 5.647 | 1.514 | 0.785 | 20.368 | 6.249 |
| 3 | 4.671 | 1.779 | 1.230 | 24.828 | 5.507 | 1.507 | 0.662 | 20.445 | 6.096 |
| 4 | 4.865 | 1.867 | 1.250 | 25.514 | 5.423 | 1.524 | 0.644 | 22.442 | 6.034 |
| 5 | 4.980 | 1.929 | 1.239 | 24.644 | 5.361 | 1.449 | 0.613 | 24.902 | 6.105 |
| 6 | 5.586 | 1.928 | 1.277 | 27.529 | 5.816 | 1.664 | 0.672 | 25.339 | 6.764 |
| 7 | 5.692 | 1.906 | 1.316 | 28.144 | 5.862 | 1.605 | 1.742 | 23.562 | 6.471 |
| 8 | 5.614 | 1.818 | 1.362 | 28.270 | 6.623 | 1.744 | 0.672 | 22.384 | 7.382 |
| 9 | 6.083 | 1.875 | 1.369 | 28.866 | 6.803 | 1.833 | 0.703 | 22.543 | 7.507 |
| 10 | 6.158 | 1.830 | 1.357 | 29.642 | 6.926 | 1.934 | 0.944 | 22.974 | 7.861 |
| 10+ | 7.179 | 1.757 | 1.391 | 28.163 | 8.660 | 2.230 | 1.137 | 21.698 | 9.895 |
| 20+ | 9.170 | 1.705 | 1.355 | 31.481 | 10.502 | 2.897 | 0.849 | 21.533 | 12.505 |
| 50+ | 9.290 | 1.599 | 1.427 | 22.894 | 9.524 | 2.862 | 0.497 | 17.623 | 12.822 |
| 100+ | 5.140 | 1.233 | 1.023 | 8.837 | 9.674 | 1 | 0.047 | 12.651 | 12 |

What we observed in the analysed sample is an increasing trend for several of the OO metrics, as long as the number of developers increases. This is especially

visible in Table 6, where the CBO, NIM and WMC average values more than double, as long as the number of developers on the Java classes increase from 1 to over 20. While for most of the metrics this might be problematic, the most prominent increasing trend is shown by the LCOM measure: the interaction of an increasing number of developers deteriorates a few of the other structural characteristics, but it has a positive effect on the cohesion of the underlying classes, thus increasing their maintainability. Table 7 summarises the findings that were observed from the sample of projects, as opposed to the guidelines expected by previous research. When the team of contributors becomes extremely large (in our sample, over 100) the classes have a higher chance to show lower values of the selected OO attributes.

It is important to note that, from the distribution of Figure 1), most classes are developed by a relatively small number of developers. It is nonetheless important to determine the relationship between large and very large teams of developers and OO structural attributes, although they represent extreme cases of the developers distribution. In section 5.4 we show in practice how a very large team of contributors has managed to keep a Java class relatively simple, from the structural point of view.

Table 7: Summary of guidelines for the selected OO metrics, and the relative observations

| OO Metric | Guideline | Observed |
|---|---|---|
| CBO | LOW | $\uparrow$ |
| DIT | LOW | $\leftrightarrow$ |
| IFANIN | LOW | $\leftrightarrow$ |
| LCOM | LOW | $\downarrow$ |
| NIM | LOW | $\uparrow$ |
| NIV | LOW | $\nearrow$ |
| NOC | LOW | $\nearrow$ |
| RFC | LOW | $\uparrow$ |
| WMC | LOW | $\uparrow$ |

From this analysis we concluded that

there is a clear effect on the structural attributes of a Java class when the number of its contributors increases

### *4.5. The Effect of Experience of Developers on OO Metrics – – $H_{0,3}$*

The analysis reported in 4.4 is repeated below, but this time considering the experience of developers as a factor in the interpretation of the results. As a reminder, we considered types of developers (Top, Middle and Bottom) based on how they worked on the codebase of each project, and how many Java files overall they created or modified. In order to avoid bias in the attribution of effort, we did not consider those commits where the amount of files touched exceeded a certain threshold (in our case, 100 Java files).

We analysed the influence of the experience in four cases: (i) when only considering the Java files worked on by Top developers; (ii) when considering a mixed team of contributions, committed mostly by top developers (e.g., $TD > MD+BD$); (iii) when considering contributions from middle and bottom developers mostly (e.g., $MD + BD > TD$); and (iv) when the top developers are not involved in any way on some specific Java file (e.g., TD = 0).

We present the analysis of these scenarios below, and Table 8 summarises the average values of each OO metric, per scenario.

#### *4.5.1. OO Metrics and Top Developers*

The results of the average for each OO metric (in relation to only Top developers) are reported in the two parts of Table 8. Every row contains the developer clusters (1 developer, 2 to 5 developers, 6 to 10 developers, more than 10 developers) of the Java files modified only by Top developers.

The metrics observed when only one (Top) developer is involved serve as the benchmark for the rest of the clusters: we observed a drop to an ideal (i.e., minimum) state when only one developer worked on the Java files. Increasing the number of developers has an impact on all metrics: in particular, the CBO, RFC and WMC metrics follow a steep growing curve that, for example, brings to $\tilde{8}$ the average value of coupling between objects.

Table 8: Growth of the OO metrics in different scenarios of developers experience)

| | Only Top Developers | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | CBO | DIT | IFANIN | LCOM | NIM | NIV | NOC | RFC | WMC |
| 1 | 4.596 | 1.756 | 1.383 | 29.697 | 5.349 | 1.368 | 0.632 | 19.917 | 5.892 |
| 2 to 5 | 4.859 | 1.850 | 1.264 | 25.491 | 5.417 | 1.453 | 0.732 | 22.406 | 5.964 |
| 6 to 10 | 5.763 | 2.019 | 1.280 | 25.450 | 5.490 | 1.458 | 1.262 | 27.786 | 6.132 |
| >10 | 7.853 | 1.890 | 1.318 | 29.250 | 7.307 | 1.985 | 2.871 | 27.466 | 8.256 |
| | TD >MD+BD | | | | | | | | |
| 1 | 4.596 | 1.756 | 1.383 | 29.697 | 5.349 | 1.368 | 0.632 | 19.917 | 5.892 |
| 2 to 5 | 4.834 | 1.833 | 1.259 | 25.685 | 5.514 | 1.479 | 0.714 | 21.809 | 6.121 |
| 6 to 10 | 5.738 | 1.885 | 1.325 | 28.195 | 6.188 | 1.707 | 0.980 | 23.956 | 6.969 |
| >10 | 7.228 | 1.761 | 1.394 | 28.522 | 8.700 | 2.209 | 1.166 | 21.824 | 9.947 |
| | MD + BD >TD | | | | | | | | |
| 1 | 5.420 | 1.787 | 1.250 | 30.292 | 5.472 | 1.774 | 0.577 | 19.055 | 6.192 |
| 2 to 5 | 4.686 | 1.554 | 1.185 | 26.262 | 5.522 | 2.065 | 0.635 | 14.687 | 6.348 |
| 6 to 10 | 4.919 | 2.290 | 1.178 | 22.235 | 6.274 | 1.577 | 0.424 | 15.583 | 6.838 |
| >10 | 5.880 | 1.610 | 1.333 | 17.900 | 7.591 | 2.219 | 0.364 | 14.452 | 8.495 |
| | TD = 0 | | | | | | | | |
| 1 | 5.420 | 1.787 | 1.250 | 30.292 | 5.472 | 1.774 | 0.577 | 19.055 | 6.192 |
| 2 | 5.749 | 1.702 | 1.209 | 29.499 | 5.514 | 2.215 | 0.472 | 20.068 | 6.598 |
| 3 | 7.790 | 1.395 | 1.086 | 26.457 | 6.667 | 2.358 | 2.370 | 25.840 | 7.741 |
| 4 | 8.688 | 2.375 | 1.188 | 37.750 | 8.313 | 3.688 | 3.063 | 31.313 | 8.688 |

*4.5.2. OO Metrics and Mixed Teams of Contributors*

We considered the scenarios of mixed teams, and a majority of top developers: the results that we obtained are mostly aligned with those by Top developers only (second part of Table 8).

When the contributions are mostly committed by Middle and Bottom developers, we observed a decrease in the value of the OO metrics (third section of Table 8); but when we considered the Java files worked on by anyone but Top developers, we observed the highest values of the sample (final section of Table 8). From the various analyses above we concluded that

less experienced developers contribute more to the decay of structural characteristics than more experienced developers

## 5. Case Studies

In this section we closely analyse 4 cases where the interaction (or lack of) between developers had an effect on the structural attributes of the underlying Java classes. We separate two case studies (sections 5.1 and 5.2) where only one developer worked on a specific class with higher-than-average structural complexity; from two further cases (sections 5.3 and 5.4) where multiple developers input code to the same Java class, also resulting in higher-than-average structural complexity.

*5.1. One Developer, High Complexity, No Maintenance*

The first case study is based on a test Java class, named *Annotations57649Test*, from the *j2objc* project[17]. It represents the Java class with the highest value of coupling between objects (CBO) in our entire sample, as seen in the following breakdown (see Table 9):

This class is a stub for a large number of further tests, and the Table above reflects how its CBO measurement is affected by the number of tests. The

[17] `https://github.com/google/j2objc`

Table 9: Structural attributes for the *Annotations57649Test* class, from the *j2objc* project

| Attribute | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 6,009 | 0 | 1 | 0 | 3 | 15 | 2 | 0 |

coupling is a result of the multiple invocations to the Retention mechanism in Java, as in the following code snippet:

```
(...)
  @Retention(RetentionPolicy.RUNTIME) @interface A0 {}
  @Retention(RetentionPolicy.RUNTIME) @interface A1 {}
  @Retention(RetentionPolicy.RUNTIME) @interface A2 {}
(...)
```

The file was added to the codebase by one of the top developers, and it never underwent any changes since its initial creation. This is because the test file belongs to a third-party project, the Android's *libcore* library, and it was deemed as functional by the developer who imported and adapted it to the *j2objc* project.

Although the class has a large structural complexity (in the form of a large CBO), further changes to this class were not needed, as long as the project evolved. This class is an example of a single-developer Java class, that encapsulates high complexity, but does not need further maintenance.

### *5.2. One Developer, High Complexity, Large Maintenance*

The second case study is based on the *aws-sdk-java* project[18], and the *AWSGlueClient* class. The class was originally created as a large, 2K lines of source code (not considering comments or blank lines), that has grown to 4K in two years. *AWSGlueClient* is a large, structurally complex class, as shown by each of the measured OO attributes (especially CBO and RFC). The latest

[18] `https://github.com/aws/aws-sdk-java`

distribution (e.g., at the time of sampling) of its structural metrics is presented in Table 10.

Table 10: Structural attributes for the *AWSGlueClient* class, from the *aws-sdk-java* project

| Attribute | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| Value | 2 | 540 | 1 | 254 | 2 | 256 | 318 | 2 | 83 |

This class underwent 33 changes since its inception, and only one GitHub developer has been in charge of its maintenance so far, for the last couple of years. On closer inspection, the file is maintained by developers of the AWS (Amazon Web Services) project, who commit under the same GitHub name (i.e., "AWS"). No other GitHub developers have worked on this Java file.

From the `git log` command, we established the revision hash of the commits where this class was modified. Through the `git reset` mechanism, we restored the *aws-sdk-java* project to each of the revisions when the *AWSGlueClient* was modified[19], then we re-evaluated the OO metrics of the project at that stage. This way, we were able to obtain the growth trend for the OO metrics of the *AWSGlueClient* class: we plotted the CBO trend in Figure 5, together with the evolution of the class in source lines of code.

A shown in the graph, the *AWSGlueClient* class has so far an unbounded growth in both lines of code, and its structural characteristics: the correlation coefficients between any of the OO metrics collected, and the SLOCs attribute is consistently above 0.9. In addition, the container Java file[20] does not contain further (inner) classes other than the *AWSGlueClient* class. This is an example of a Java class that constantly grows its structural complexity, but does not benefit from other developers' work.

[19]For example, the `git reset --hard 6cd91c1f6a4cabea5b1f877e5204247e60069f89` command will restore the *aws-sdk-java* project to the state when the *AWSGlueClient* class was first introduced.

[20]Its full path is `aws-java-sdk-glue/src/main/java/com/amazonaws/services/glue/AWSGlueClient.java`

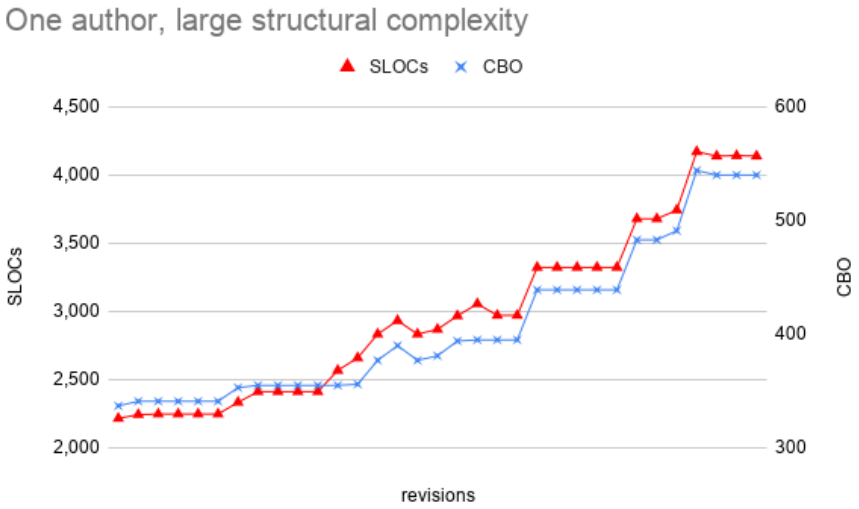



Figure 5: Growth of lines of code and CBO for the *AWSGlueClient* class

### *5.3. Many Developers, Large Maintenance, High complexity*

The third case study is based on the *cassandra* project[21], and specifically about the main class contained in the file `StorageService.java`. The latest OO metrics that we collected for this class are displayed in Table 11. What is also listed in the first column of the Table is the cumulative number of authors that made changes on the class, since its inception: we counted up to 116 distinct author IDs that made changes to this class throughout its growth, and up until our sampling date.

Similarly to what was noted in the case study of section 5.2 above, this class shows the attributes of high structural complexity (e.g., CBO=150, NIM=341) while remaining relatively simple from the hierarchical point of view (e.g., NOC=0, DIT=2).

This class underwent some 1,700 revisions in its evolution: similarly to what was done for the *AWSGlueClient* class above, we restored the project to each

[21] `https://github.com/apache/cassandra`

Table 11: Structural attributes for the *StorageService* class, from the *cassandra* project

| Authors | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| 116 | 3 | 150 | 0 | 341 | 28 | 348 | 348 | 2 | 95 |

intermediate revision and recorded its structural characteristics at those revisions. Figure 6 shows the growth of the CBO attribute, alongside the number of source lines of code, against the cumulative number of developers.

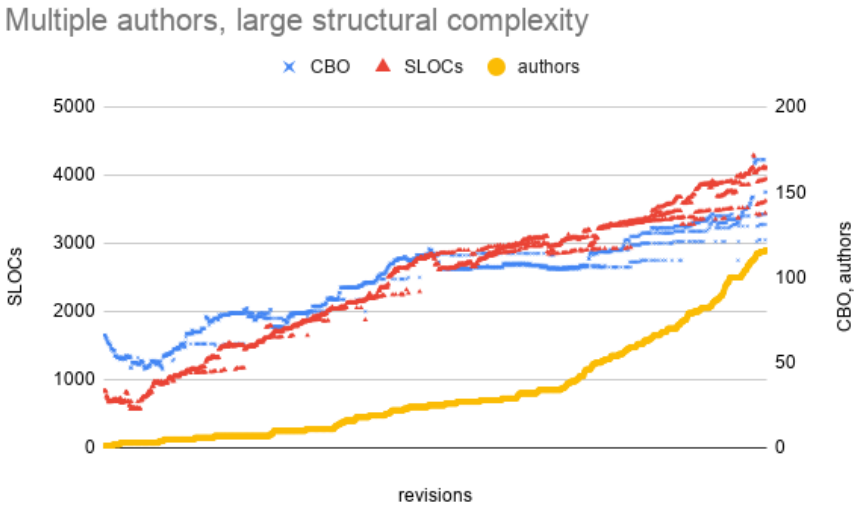



Figure 6: Growth of number of authors, lines of code and CBO (*StorageService* class)

The effects of multiple authors, and the basic difference with the *AWSGlueClient* class, is visible in the number of different branches of development that this class benefits from (as shown in the parallel lines of SLOC and CBO data from the Figure). The influence of multiple developers is also visible in the number of inner classes that have grown inside the main one: in its inception, only one further inner class was present (the static class *BootstrapInitiateDoneVerbHandler*), then two inner classes were developed beside the *StorageService* one, while the latest revisions revert to a single inner class (e.g., *RangeRelocator*).

Even so, the main class had a five-fold growth in terms of size, and three-fold

in terms of CBO; the cumulative number of developers is positively correlated with the size of the Java file, but the structural complexity does not seem to be bounded.

### *5.4. Many Developers, Large Maintenance, Low Complexity*

The last case study that we propose is based on the *teammates* project. We focused on the *Const* class, that, in our sample, has the largest number of distinct authors working on the same class (i.e., 133).

We collected the OO attributes in the first and last revisions, for comparison (see Table 12). In both revisions, the OO metrics are at their lowest possible values, while the total number of lines of code doubles from 552 (with 432 source lines of code) to 1,131 (857 source lines of code).

Table 12: Structural attributes for the *Const* class, from the *teammates* project at its initial revision (first row) and latest revision (second row)

| Authors | IFANIN | CBO | NOC | NIM | NIV | WMC | RFC | DIT | LCOM |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 2 | 0 | 2 | 2 | 1 | 100 |
| 133 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 100 |

We also observed that the class underwent 854 revisions: for each we noted the source lines of code, and the number of cumulative developers that worked on the class. Same as above, we evaluated the structural attributes of the class at each revision, and Figure 7 plots the cumulative number of developers, alongside the size of the class.

What we also plotted in Figure 7 is the growth of (the number of) inner classes (from the initial 8 to 24) that have been (and are) part of the `Const.java` source file, alongside the main *Const* class.

What we observed in this case study and from the plot (i.e., Figure 7) is that there is virtually no correlation between the growth in OO metrics or the class size and the number of its authors. The structural growth of the source file is achieved via the number of newly added inner classes, each remaining

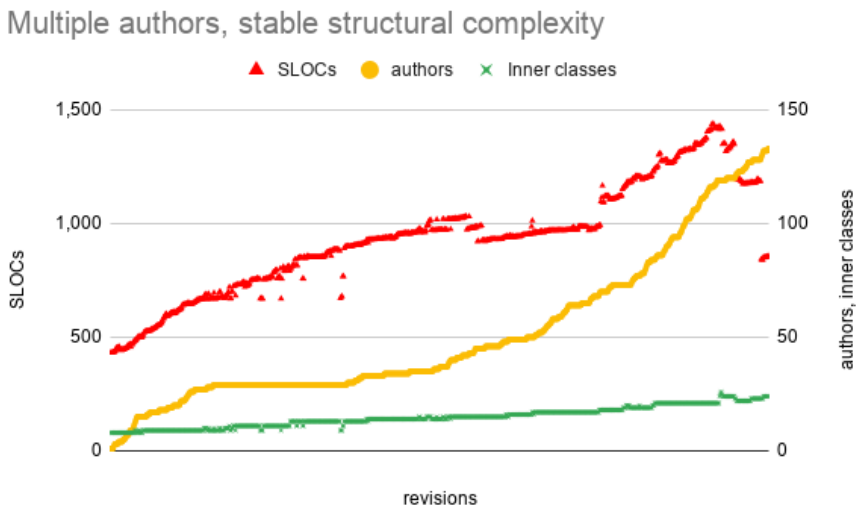



Figure 7: Multiple Developers Making Commits for and with the Same Class

minimally complex, from the structural point of view. Observing the SLOC and inner classes plots in Figure 7, it can be observed that when there is a drop in the SLOC (red), there is also a drop in the number of inner classes (green). On the other hand, a drop in the SLOC plot (red) does not imply a drop in the number of authors plot (yellow).

## 6. Discussion and Threats to Validity

In this section we discuss the results from our findings, dividing them into two parts: the correlation levels between OO metrics (section 6.1), and the relationship between individual OO metrics and number of developers (section 6.2).

### *6.1. Literature Findings on Correlations*

Considering the overall number of classes contained in the sampled projects (474,197), we observed some strong correlations in action. Past research has already established correlations between OO metrics, and at varying levels of

strength. What we discuss below is how our results could be used (i) to complement existing, established results from the literature, and (ii) to shed new light towards potentially new explanations for the results obtained in past research.

As one of the first research studies attempting to find correlations between OO metrics, Systa *et al.* [49] found strong correlations between RFC and other attributes: WMC, LCOM, DIT and CBO. The results from our sample of Java classes concur with those results: all of those correlations (plus a few more) were found to be significant. What we observed is that these correlation levels are not always stable for any number of developers. For instance, the RFC v WMC relationship shows an overall *moderate* correlation coefficient, but it becomes *large* when considering only the classes developed by only one developer, or the classes where over 10 developers input their code. We posit the following: what is reported by Systa *et al.* [49] might be due to the specificity of the analysed system (the FUJABA project, developed at the host institution), and the number of developers involved in that project (less than 5).

As a second example of correlation results reported in the literature, Olague *et al.* [50] examined 6 versions of Rhino, an open source implementation of JavaScript. The authors reported that the versions of Rhino analysed in the paper have been developed by 3 programmers. Their correlation study indicates that WMC strongly correlates with RFC, CBO and LCOM. This is consistent with our results, when considering the clusters of developers, and Rhino being in the [2 - 5] bracket. The authors further state that:

'*Rhino's CK-RFC metric correlated more strongly with CK-CBO and CK-LCOM than occurred in previous studies*'

They also observe that:

'*...(the) primary differences between this study and previous studies were NOC in this study had either no correlation or minor correlation with CBO (previous studies showed no correlation) and CBO had a moderate to large correlation with LCOM (previous studies showed a small or no correlation)*'

As above, the differences observed by the authors can be ascribed to the fact that Rhino belongs to the [2 - 5] developer bracket: the other cited studies

have performed such correlation analysis without taking into consideration the number of developers involved.

As a further example of a past correlation study reported in the literature, the work reported by Gyimothy *et al.* [51] used Mozilla as a case study to evaluate the correlation between the C&K metrics: the authors compared their results with what found by Basili *et al.* [52], and observed a few differences, in particular a higher correlation between WMC and RFC, as well as between WMC and CBO. This is consistent with what we report in Table 5: the system studied by Gyimothy *et al.* [51] is a large Open Source system (i.e., Mozilla) whose classes are modified by a large number of developers. On the other hand, the systems studied by Basili *et al.* [52] were student projects: the set of results proposed by the authors [52] are more in line with the correlations found in presence of only one developer.

Subramanyam and Krishna [30] reported that a high CBO combined with a high DIT of classes has a higher effect on software defects in C++ compared to Java. However, the authors acknowledge that the sample was skewed, with a small number of classes with high values of DIT.

### *6.2. Trends in OO metrics and developers*

The trends shown in Table 7 demonstrate that most of the OO metrics studied increase as the number of developers working on a class increases. While we might reasonably expect this trend as the number of developers increases (since system LOC will also generally increase correspondingly), there are a number of implications for rises in the value of certain metrics evident Table 7 and these are worth exploring.

#### *6.2.1. CBO*

Software maintenance research generally emphasizes the need to keep metrics low. For example, it has been shown that an increase in the CBO for a class can lead to an increase in the required maintenance effort, defects and a reduction in the reusability [53, 30]. This is because the higher the CBO of a class, the

more sensitive the software (i.e., other coupled classes) will be to changes made to that class. Results from Table 7 show that CBO rises significantly as number of developers increases; in terms of both mean and median values, there appears to be a more dramatic rise after five developers in each case. Perhaps it is at this level (of developers) that the complexity of the system reaches a tipping point. In other words, coupling is added indiscriminately through a lack of system understanding and poor communication, the net result of which is a large technical debt [54, 55].

### *6.2.2. DIT*

In terms of DIT, the deeper a class is in the hierarchy (i.e., has a high DIT value), the greater the number of methods it is likely to have inherited from parent classes, making its behaviour more complex to predict. On the other hand, classes with a small DIT have much potential for reuse. Table 7 shows that the DIT values remain relatively static as the number of developers increases. This is not entirely unexpected. A number of previous studies have shown that DIT values tend to be generally low and that, if anything, inheritance hierarchies will tend to collapse over time (becoming shallower) rather than deeper [56, 52, 57]. Typically, this leads to systems with low median DIT values of one or two, as Table 7 shows. From the same figure, we actually see a small fall in the DIT value as number of developers increases. A number of suggestions can be put forward for why DIT exhibits this trends. There is some research to show that beyond a certain level of inheritance, developer comprehension becomes lowered [58, 59]. Another way of describing this is in terms of the cognitive load on developers. While the original intention of inheritance was to promote reuse through relatively deep levels of inheritance (and developers would therefore always strive to add depth to inheritance hierarchies to achieve this), it seems developers prefer simplicity of shallow structures instead.

### *6.2.3. NOC*

The NOC values are also interesting and show two peaks (at developer 4 and >10), before returning to a low level (approximately 1). We can only speculate as to why these peaks occurred (we note that the median values remain static). One suggestion is that at the point in the system there was a significant re-engineering or refactoring effort which collapsed the overall hierarchy temporarily. This would have had little effect on the other OO metrics, but would significantly increase the mean NOC values as shown. It is remarkable that the median NIV value also jumps at this point (0 to 1), further pointing to possible large-scale merging of classes at the same time as the hierarchy being collapsed; again this is speculation only, but is often an activity applied to systems when they show signs of decay.

### *6.2.4. LCOM*

As well as CBO, the LCOM metric also shows a steep rise in both mean and median values; for median LCOM values, this is particularly noticeable after the 3 developer level. A raised LCOM implies that a class is becoming increasingly fragmented and losing much of its functional coherence. The LCOM value is influenced by many factors and from the data it is difficult to pinpoint why the LCOM should rise so steadily. One suggestion, however, is that as the number of methods in a class rises (according to WMC), the distribution of instance variables around the classes of the system become more thinly spread. The result of this is a lowering of cohesion (and consequent rise in the LCOM value). In other words, if the responsibilities of a class become less related, then class cohesion will inevitably suffer as a result.

### *6.2.5. Summary Observations*

One observation that seems to hold true is that as more developers are added, a number of key OO metrics measuring coupling, size and cohesion worsen. It would be easy to say that this is inevitable due to systems naturally decaying [60]. We cannot ignore the fact that more developers usually means

more room for human error (through communication and misunderstanding) implies. Brooks' Law may have played a large part in what we observed [61].

Another observation relates to the link between number of developers and the OO metrics used. One criticism of the work is that we do not need to study the link between developer numbers and metrics. Clearly however, there are significant changes in some of the OO metrics as number of developers increases. We would largely expect that (and for those metrics to worsen). However, it is the specific interesting cases for example of CBO, NOC and LCOM which highlights the synergy between the number of developers and the metrics and which makes this analysis worthwhile.

### *6.3. Repercussions on Software Maintenance*

The evaluation of the hypotheses H1, H2 and H3 above pointed to an increase in values for most of the structural metrics that were examined: when more developers have worked on the same Java files, their structure 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=989965 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, while still avoiding the commits where more than 100 Java files were modified at the same time.

The results are found in Table 13 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.

Table 13: Average and median number of additional commits per cluster of developers, and considering experience as a factor

| | Overall sample | | | | |
|---|---|---|---|---|---|
| Dev. clusters | all | 1 | 2 to 5 | 6 to 10 | 10+ |
| Further commits (AVG) | 8.11 | 2.95 | 11.47 | 26.26 | 47.48 |
| Further commits (MED) | 4 | 2 | 9 | 22 | 42 |

| | 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 |

When considering all the Java classes in our sample (section "Overall sample" in Table 13) we observed that, on average, the classes modified by one developer are those that needed the least further maintenance (less than 3 additional commits, on average; and 2 further commits as a median value). When 2 to 5 developers have worked on a Java class, the additional commits become more than 11 on average, and 3 as a median. The additional maintenance becomes much more visible in the "6 to 10" 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 22 additional commits; in the latter a median of 42 additional commits.

The same trend is visible when only the Top developers (section "ONLY TOP DEVELOPERS" in Table 13) 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. Classes with an increasing number of Top developers tend to degrade structurally (see section 4.5.1) but they do not need

as much further maintenance as compared to classes with mixed Top, Middle and Bottom developers.

This second finding is confirmed by the analysis of classes whereby the number of Middle and Bottom developers that have modified the class is higher than the number of top developers that have modified the class (section "$MD+BD > TD$" or the last three rows in Table 13). The development of Java classes by developers with mixed experience levels shows a visible effect on their maintenance, requiring a lot more further commits (both in average and median) than when the only Top developers work on the code. For example when looking at the classes modified by 6 to 10 developers in section "$MD + BD > TD$" or the last three rows in Table 13 compared to section "ONLY TOP DEVELOPERS" in Table 13, the average number of further changes needed is much higher in the former (16.82) compared to the later (11.56).

As a way of an example, Figures 8 and 9 show two recurring types of maintenance (both examples are taken from the *AndroidAnnotations*[22] project): the former where a majority of Top developers was active in the evolution of the Java file, and the latter where more Middle and Bottom developers have been modifying the class, as opposed to the Top developers. Each change that the two files underwent was assigned to a developer, and each developer was assigned to one of the TD, MD or BD categories.

The pattern observed in Figure 8 is sufficiently recurrent in other Java classes where most developers are in the TD category. The maintenance is relatively regular and evenly scattered, although a gap of over a year separates a first and a second phase of maintenance (highlighted by different colours). On the other hand, the pattern observed in Figure 9 shows a heavy involvement of MD and BD types of developers, but only for roughly half of its life-cycle. In the second part of its maintenance, the support of Middle and Bottom becomes virtually null, whilst the maintenance of this second half is carried forward by Top developers.

[22] `https://github.com/androidannotations`

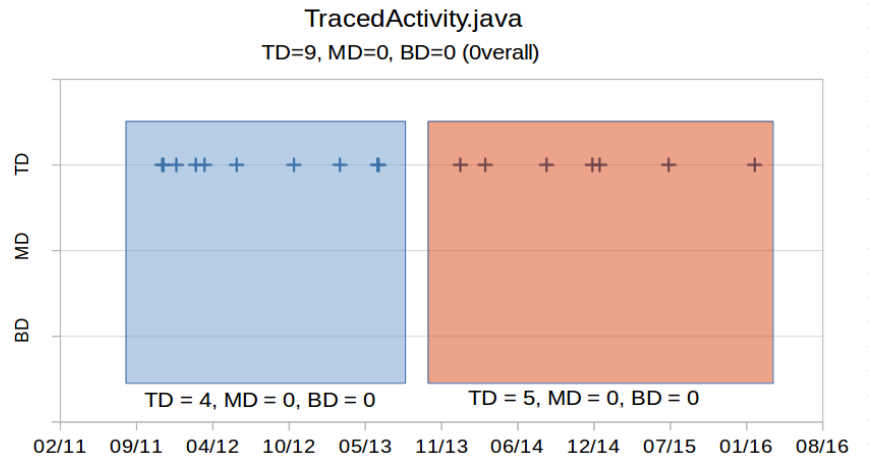



Figure 8: Maintenance for the `TracedActivity.java` file

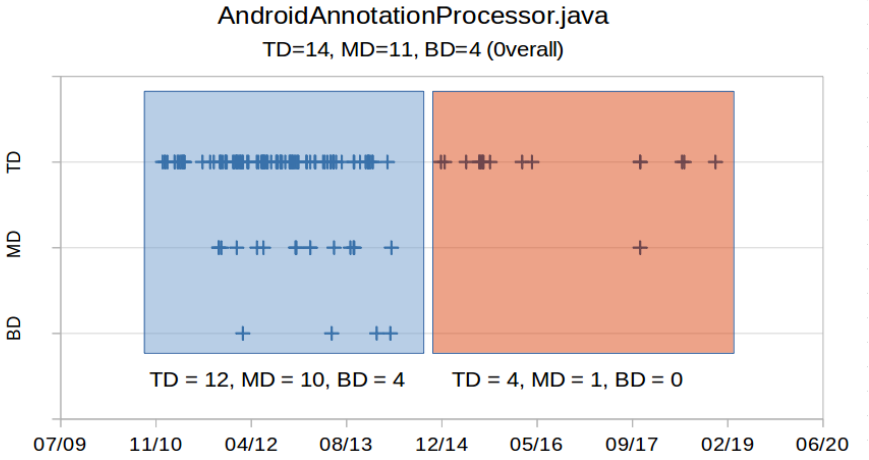



Figure 9: Maintenance for the `AndroidAnnotationProcessor.java` file. Ticks indicate individual changes

### *6.4. Threats to Validity*

In this Section we present the *external*, *internal* and *construct* threats to validity of this study.

*External validity* – This paper presents the results of an empirical analysis that should be applicable to all OSS projects. We cannot generalize our findings

on any other sample of OSS projects, or from any other repository. Nonetheless, in order to make the findings from our study more generalisable and representative of OSS projects, we have carried out our analysis on a significant sample of projects, with different sizes and number of developers.

*Internal validity* – Another threat to the study is a possibility of a migration of projects between open source forges. We acknowledge that some of the projects might have been migrated from one repository to another. This could mean that there are subtle inaccuracies in the number of developers observed to have accessed a class. However, to mitigate this threat we examined the initial commit logs of the studied projects by means of SQL queries on the commit logs and parsed these to identify whether the developers' commit messages indicate any migrations from another repository.

We identified as initial commit messages: "Create README.md", "Hello World!", "Initial commit", "Initial release. As a proof of concept, it already works in javadoc and eclipse!" and "first commit". Second commit messages include "Updated README.md", "Updated .gitignore", "Make sure our Guice-Container is a singleton", "Formatted readme" and "Project description".

One other internal validity threat alluded to in the previous section is the interplay between size of the system and number of developers. A criticism that could be levelled at the study is that the C&K metric values are only a reflection of the size of the systems as they evolve and not due to the increasing number of developers (i.e., size is a confounding factor). In defence of this threat, we accept that as the number of developers increases, the size of the system will grow (notwithstanding Brooks Law and that the C&K metrics will tend to worsen; however, the size and number of developers are effectively surrogates for each other and do not invalidate the analysis.

*Construct validity* – To assess the presence of a linear correlation between the software metrics as well as the number of developers accessing a class we adopted the Spearman's rank correlation coefficient. This is because it does not assume a normal distribution. The test has its disadvantages: it takes into consideration the ranked order of the attributes and not the values themselves.

In other words, as long as the order of the C&K metrics remain the same the resulting coefficient will stay the same. As such, the results rely upon the time or period at which the studied sample has been collected.

Additionally, the scope of our sample of projects was limited to open-source software projects written in the Java programming language (object-oriented), thus we encourage investigating commercial projects for our results to be inclusive and completely validated.

It is also important to note that the study that we proposed is a quantitative evaluation of internal attributes of Java systems. We tried to triangulate these findings with the case studies presented in Section 5. Further insights could get unearthed by conducting a mixed-approach research [62], including questionnaires and interviews with managers and developers. We believe that the scale of our research does not lend well to that type of research though: mixed-research methods are particularly efficient when a limited number of systems are analysed, and more precise research questions are formulated regarding specific systems.

Finally, our study used (i) test and non-test classes, (ii) small, medium and large systems size and (iii) experience of developers as three possible controlling factors for the trend of the CK metrics. We acknowledge that many other factors could play a role in these patterns: the type of systems, above all, should be further investigated to unearth relationships between structural characteristics and developers [63]. Also, the level of involvement of commercial companies in open source systems (the so-called *hybrid* OSS systems [64, 65]) would have a role to play. Company-specific development standards would become more visible in company-driven OSS projects: in this way, it would also be possible to compare them to volunteer-based OSS projects.

## 7. Related Work

Section 2 provides a background review of the studied OO metrics. These metrics have also been evaluated with nine criteria for OO software complex-

ity metrics [15]. They are: (1) non-coarseness, (2) non-uniqueness, (3) design details importance, (4) monotonicity, (5) non-equivalence of interaction, (6) interaction increases complexity, (7) complexity in response to the order of statements and the interaction among statements, (8) renaming and equality in complexity (9) program growth and increased complexity.

The WMC metric meets the first and second metric properties defined by Weyuker [15] as not every class can have the same number of methods but some classes can. In addition, the functionality of a class does not define the number of methods the class can contain. Therefore, the WMC metric also satisfies the third property. The choice of number of methods is a design decision and is not based on functionality.

The DIT metric satisfies property 1 given that the depth of inheritance of a leaf class is always greater than that of the root class. Furthermore, there will also exist at least two classes with the same depth of inheritance since every tree has at least some nodes with siblings. As such the DIT metric also satisfies property 2.

However, the DIT metric fails to satisfy property 4 in cases where two classes are in a parent-descendant relationship. This is because the distance from the root of a parent cannot be greater than one of its children. It is noteworthy that not satisfying property 6 may not be an essential aspect of OO design. This is because, developers have identified that the division of classes into more classes can increase complexity too (in terms of memory management and runtime detection of errors when there are more classes to deal with). On the other hand, developers can make use of the WMC, DIT and NOC metrics to check if an OO software is getting "top heavy" (i.e., too many classes at the root level declaring many methods) or use the RFC and CBO metrics to check whether there are unneeded interconnections between various parts of the application.

In contrast to Weyuker, Ma *et al.* [66] categorized software complexity metrics based on their purpose or the general software properties they measure and their limitations. Their categorization of the CK metrics are as follows: *Inheritance* (DIT, NOC), *Coupling* (CBO), *Collaboration* (RFC), *Cohesion* (LCOM),

*Complexity* (WMC).

### *7.1. Comparison of software attributes*

Gyimothy *et al.* [51] performed a comparison of the CK metrics and it was discovered that there is no linear relationship between NOC and the other metrics. In addition, DIT only correlated with RFC. Another notable correlation is that of WMC and LCOM. Finally, LOC correlated with WMC, RFC, CBO, and LCOM but not DIT and NOC. This implies that the degree of cohesiveness of a class[23] can determine its number of lines of code (LOC). Counsell *et al.* [67] state that the LCOM metrics is an implementation metric required earlier in the development process (at design time).

Similarly, in a different study on fault prediction with a different data set, Zhou and Leung [68] identified a significant linear correlation at the 0.01 level between DIT and RFC. In addition, SLOC correlated with WMC, RFC, CBO, DIT and LCOM but NOC. Though a low correlation with LCOM (0.24) and DIT (0.35).

It can be inferred from both studies performed with distinct case studies or data sets, that NOC and lines of code of a class have no relationship. Furthermore, the DIT and NOC metrics for classes in studies of OO software are usually low. This indicates that deep inheritance values are not used significantly in OO software. On the other hand, a high CBO is evident among classes which means that the dependencies between classes which is not caused by inheritance is high [68, 52, 69].

Table 14 summarizes the known C&K metric relationships identified statistically using correlation measurements as well as key findings regarding the metrics. The first column shows the metric name, the second column lists other correlated metrics, the third column summarizes some key findings about the metric and relationships and the fourth and last column shows the studies that

[23] A cohesive class is one in which the same instance variables appear in most or all of the methods.

Table 14: Summary of OO Metrics Relationship and Operationalisation

| Metric | Metric Links | Findings | Study |
|---|---|---|---|
| LCOM | LOC | LOC is correlated with LCOM | [51] |
| | SLOC | SLOC correlated with LCOM | [68] |
| DIT | SLOC | SLOC correlated with DIT | [68] |
| CBO | LOC | LOC correlated with CBO | [51] |
| | CBO | SLOC correlated with CBO | [68] |
| NOC | | The higher the number of children a class has, the greater its reuse since inheritance is a form of reuse. However, a higher inheritance means that the class design will be complex to test | [9]; [31] |
| RFC | DIT; LOC | DIT and LOC correlated with CBO | [51] |
| | SLOC | SLOC correlated with RFC | [68] |
| | DIT | High CBO combined with a high DIT produces a higher effect on defects in OO software classes | [30] |
| | | Excessive coupling between classes is detrimental to modular design and lowers the chances of reuse | [9] |
| | | CBO increases complexity of the system and adversely affects the quality factors such as maintainability, testability and reusability | [32] |
| | | A measure of coupling is important to determine how complex the testing of various parts of a design are likely to be | [19] |
| WMC | LCOM | WMC correlated with LCOM | [51] |
| | SLOC | SLOC correlated with WMC | [68] |
| | | A high value of WMC could result in a high number of software faults as classes with a high number of methods are more difficult to reuse and maintain. Increasing the average of WMC also elevates complexity but lowers quality | [29] |
| | | The larger the number of methods in a class, the greater the potential impact on children, since children will inherit all the methods defined in the class | [9] |

have statistically proven the metric pair correlations and other findings.

In a survey of metrics available for UML diagrams [70], only three (WMC, DIT and NOC) of the CK metrics were available for UML diagrams and applied equally to models and code [71]. These metrics can be used to measure design complexity in relation to their impact on external quality attributes (e.g., maintainability and reusability).

The DIT metric is seen as a length measure while the NOC metric is a size measure. According to Briand *et al.* [72] DIT and NOC both have opposite effects on fault detection. The higher the DIT metric, the greater the chances of detecting a fault in a class. On the other hand, the higher the NOC metrics, the lower the chances of fault detection [70]. However, deeper inheritance hierarchies did not speed up maintenance, so DIT in itself is not an important factor for maintenance effort. Similarly, Aggarwal *et al.* [36] hypothesized that a class with less cohesion is more likely to be fault prone than a class with high cohesion. While a class with more depth in its inheritance tree is more likely to be fault prone than a class with less depth in its inheritance tree.

### *7.2. Software attributes and collaborative development*

Bird *et al.* [5] demonstrated that code ownership has a relationship with software defects considering size, churn and complexity metrics extracted from the Windows Vista and Windows 7 software projects. The authors recommended that changes made by minor contributors should be reviewed with more effort than changes made by major contributors or developers who are experienced with the source code for a component to minimise defects. This is as commits with large number of files or changes have been associated with less useful reviews for the author of the change [73]. Similarly, our results have shown in terms of number of developers that OO quality metrics such as DIT and NOC are correlated with the number of developers of a class. In addition, we analysed developer experience per project based on the number of Java files committed and identified that when the count of developers that have worked on a class increases or the count of the less experienced developers surpasses the expe-

rienced developers working on a class, its structure degrades and more future maintenance is required.

Understanding component and task coupling would, for example, allow organisations to be better informed about the need for more effective coordination (and resource allocation) [27], team co-location and code ownership [74]. Herbsleb proposed the need to measure architectural and organisational fit as well as the need for tactics to better adjust an organization to the software architecture, or the architecture to the organization [75]. This is because there is little knowledge on the coordination and communication requirements that architectural decisions impose on teams. For example, considering the link between task dependency and component coupling/dependencies, decoupling components might or might not decouple tasks [76]. In addition, adding an intermediary where the most difficult dependencies are semantic rather than syntactic may in fact make the task coordination problem harder.

In a study on two large and mature software projects (Firefox and Eclipse) by Parnin *et al.* [77], the researchers investigated the effects of distributed development (in terms of organisationally and geographically distributed teams) on software quality. With regards to Firefox, components that are geographically distributed were found to be larger and complex with more contributors. On the other hand, Eclipse showed a low geographic distribution at the component level (i.e., almost every component is developed largely in one location). Generally, geographically distributed components had more defects though the effects of distribution lessen in later releases. While organisationally distributed components had less defects. In a different study, organizational metrics when applied to data from Windows Vista were able to statistically and significantly predict failure-proneness [26]. Prediction model performance metrics (precision and recall) were higher when the prediction of failure-prone binaries compared to using traditional metrics like churn, complexity, coverage, dependencies, including pre-release bug measures.

According to Matsumoto *et al.* [6] the injection of faults in software does not solely depend on attributes of the source code, but also on attributes of

the developers involved in software projects. The authors investigated the relationship between developer attributes (*e.g.,* number of code churns made by each developer, the number of commitments made by each developer and the number of developers for each module) and the number of faults in software. The developer attributes were also evaluated for performance improvements of fault prediction models and the Eclipse project was used as a case study. Their results revealed that modules touched by multiple developers contained more faults and developer-based metrics improved the performance fault prediction models. Differently from their study on fault prediction, we have investigated the attributes of classes in relation to developers and identified that the correlation between OO software metrics is influenced by the number of developers that have touched a class. When considering developer experience, we also identified that the experience of developers touching a class plays a role in the complexity of the class.

Similarly to Matsumoto *et al.* [7], Ostrand *et al.* [7] investigated whether files modified by an individual developer consistently contain either more or fewer faults than the average of all files in the system with the aim of determining whether the information about which particular developer modified a file is able to improve defect predictions. The authors also evaluated the use of counts of the number of developers who modified a file as predictors of the files' future faultiness. Their study confirmed that counts of the cumulative number of different developers changing a file over its lifetime can help to improve fault predictions, but only by a small amount. Differently to this study, the authors did not investigate the experience of developers in their prediction models. In addition, we identified that when the count of developers that have worked on a class increases, its structure degrades and more future maintenance is required.

In an earlier study by Mockus and Weiss [78] the relationship between change quality and four different developer variables, mainly evaluating developers experience is used in a logistic regression model to predict whether a change leads to a software failure using a telephone switching system as a case study. The four developer variables used included: a measure of a developer's overall ex-

perience (the number of changes made to files in a project before a change in focus), developers recent experience (recent changes are weighted more than older changes), and their experience with a specific subsystem (changes made to files within a subsystem that a change touches), as well as the number of developers who made modifications to satisfy a change request. Their results showed that the only statistically significant variable was the one that measured the developer's overall experience level. In this study, we have analysed developer experience per project based on the number of Java files committed per developer and found that developers overall experience in a project has a role to play in the overall structural quality of classes they work on.

## 8. Conclusion and Future Work

This paper proposed an extensive study on the relationship between a selection of structural OO metrics and the number of distinct developers who add, delete or modify parts of Java classes.

We collected three sets of results from our analysis. First, we found that the OO metrics of a class are correlated to the number of its contributors, but not to its size in SLOCs. As long as more developers work on the same code, their correlation with other metrics increases. This result helps to put in context past literature findings: the stronger or weaker correlations found by researchers depend on the number of developers who worked on the code.

Second, we found that the experience of developers plays a role: the more inexperienced developers have a visible effect on the structural characteristics of the code that they work on, degrading it more as compared to when only experienced developers commit to Java files.

Third, we observed that the degradation in OO metrics is linked to an increase in further maintenance: when more developers work on the same Java code, its structure degrades AND the number of further commits needed will increase. This is even more visible when less experienced developers have worked (or still work) on the code itself.

We believe that these results open a new research area for the maintainability of software. Further work could be directed towards the effectiveness of repeated partnerships between developers, especially when experienced developers are found in pairs in several projects. Also, our set of findings could be linked to the on-boarding activities of community software projects, especially for those developers who belong to the Middle or Bottom layers of collaborators.

## REFERENCES


[1] I. Steinmacher, M. A. Gerosa, How to support newcomers onboarding to open source software projects, in: L. Corral, A. Sillitti, G. Succi, J. Vlasenko, A. I. Wasserman (Eds.), Open Source Software: Mobile Open Source Technologies, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 199–201.

[2] B. Lin, G. Robles, A. Serebrenik, Developer turnover in global, industrial open source projects: Insights from applying survival analysis, in: 2017 IEEE 12th International Conference on Global Software Engineering (ICGSE), IEEE, 2017, pp. 66–75.

[3] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining github, in: Proceedings of the 11th working conference on mining software repositories, ACM, 2014, pp. 92–101.

[4] B. Norick, J. Krohn, E. Howard, B. Welna, C. Izurieta, Effects of the number of developers on code quality in open source software: A case study, 2010. doi:10.1145/1852786.1852864.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, P. Devanbu, Don't touch my code!: examining the effects of ownership on software quality, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 4–14.

[6] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, M. Nakamura, An analysis of developer metrics for fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, 2010, p. 18.

[7] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Programmer-based fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, 2010, p. 19.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus, Does code decay? assessing the evidence from change management data, IEEE Transactions on Software Engineering 27 (1) (2001) 1–12.

[9] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[10] B. Fluri, M. Wursch, H. C. Gall, Do code and comments co-evolve? on the relation between source code and comment changes, in: 14th Working Conference on Reverse Engineering (WCRE 2007), IEEE, 2007, pp. 70–79.

[11] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, H. Rajan, A study of repetitiveness of code changes in software evolution, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2013, pp. 180–190.

[12] M. Foucault, J.-R. Falleri, X. Blanc, Code ownership in open-source software, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2014, p. 39.

[13] L. C. Briand, J. W. Daly, J. K. Wüst, A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on software Engineering (1) (1999) 91–121.

[14] B. Kitchenham, Whats up with software metrics?–a preliminary mapping study, Journal of systems and software 83 (1) (2010) 37–51.

[15] E. J. Weyuker, Evaluating software complexity measures, IEEE transactions on Software Engineering 14 (9) (1988) 1357–1365.

[16] J. C. Cherniavsky, C. H. Smith, On weyuker's axioms for software complexity measures, IEEE Transactions on Software Engineering 17 (6) (1991) 636–638.

[17] M. Shepperd, D. Ince, Derivation and Validation of Software Metrics. International Series of Monographs on Computer Science, Oxford University Press, 1993.

[18] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of systems and software 23 (2) (1993) 111–122.

[19] G. A. Oliva, M. A. Gerosa, On the interplay between structural and logical dependencies in open-source software, in: Software Engineering (SBES), 2011 25th Brazilian Symposium on, IEEE, 2011, pp. 144–153.

[20] A. Chhikara, R. Chhillar, S. Khatri, Evaluating the impact of different types of inheritance on the object oriented software metrics, International Journal of Enterprise Computing and Business Systems 1 (2) (2011) 1–7.

[21] S. Counsell, E. Mendes, S. Swift, Comprehension of object-oriented software cohesion: the empirical quagmire, in: Proceedings of the $10^{th}$ International Workshop on Program Comprehension (IWPC), Citeseer, 2002.

[22] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, 2010, pp. 31–41.

[23] K. El Emam, W. Melo, J. C. Machado, The prediction of faulty classes using object-oriented design metrics, Journal of Systems and Software 56 (1) (2001) 63–75.

[24] D. Radjenović, M. Heričko, R. Torkar, A. Živkovič, Software fault prediction metrics: A systematic literature review, Information and Software Technology 55 (8) (2013) 1397–1418.

[25] L. F. Capretz, J. Xu, An empirical validation of object-oriented design metrics for fault prediction, Journal of Computer Science 4 (7) (2008) 571.

[26] N. Nagappan, B. Murphy, V. Basili, The influence of organizational structure on software quality, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 2008, pp. 521–530.

[27] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 2008, pp. 531–540.

[28] M. Bunge, Treatise on basic philosophy: Ontology I: the furniture of the world, Vol. 3, Springer Science & Business Media, 1977.

[29] D. K. Srivastava, A. Singh, Classification of technical and management metrics in object oriented software engineering, in: Proceedings of International Conference on Communication and Networks, Springer, 2017, pp. 277–286.

[30] R. Subramanyam, M. S. Krishnan, Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects, IEEE Transactions on software engineering 29 (4) (2003) 297–310.

[31] S. Khalid, S. Zehra, F. Arif, Analysis of object oriented complexity and testability using object oriented design metrics, in: Proceedings of the 2010 National Software Engineering Conference, ACM, 2010, p. 4.

[32] U. L. Kulkarni, Y. Kalshetty, V. G. Arde, Validation of ck metrics for object oriented design measurement, in: Emerging Trends in Engineering and Technology (ICETET), 2010 3rd International Conference on, IEEE, 2010, pp. 646–651.

[33] G. Bavota, A. De Lucia, R. Oliveto, Identifying extract class refactoring opportunities using structural and semantic cohesion measures, Journal of Systems and Software 84 (3) (2011) 397–414.

[34] N. Ajienka, A. Capiluppi, S. Counsell, 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, 2017, pp. 141–150.

[35] A. Marcus, D. Poshyvanyk, R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE Transactions on Software Engineering 34 (2) (2008) 287–300.

[36] K. Aggarwal, Y. Singh, A. Kaur, R. Malhotra, Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study, Software process: Improvement and practice 14 (1) (2009) 39–62.

[37] M. Lorenz, J. Kidd, Object-oriented software metrics, Vol. 131, Prentice Hall Englewood Cliffs, 1994.

[38] C. Van Koten, A. Gray, An application of bayesian network for predicting object-oriented software maintainability, Information and Software Technology 48 (1) (2006) 59–67.

[39] G. Destefanis, S. Counsell, G. Concas, R. Tonelli, Software metrics in agile software: An empirical study, in: International Conference on Agile Software Development, Springer, 2014, pp. 157–170.

[40] F. Zhang, A. Mockus, Y. Zou, F. Khomh, A. E. Hassan, How does context affect the distribution of software maintainability metrics?, in: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE, 2013, pp. 350–359.

[41] A. Marcus, D. Poshyvanyk, The conceptual cohesion of classes, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, 2005, pp. 133–142.

[42] L. Yu, Understanding component co-evolution with a study on linux, Empirical Software Engineering 12 (2) (2007) 123–141.

[43] R. R. Pagano, Understanding statistics in the behavioral sciences, 6th Edition, Wadsworth-Thomson Learning, Australia;United Kingdom;, 2001.

[44] A. P. Field, Discovering statistics using SPSS: and sex and drugs and rock 'n' roll, 3rd Edition, SAGE, London;Los Angeles;, 2009.

[45] M. A. Cusumano, How microsoft makes large teams work like small teams, MIT Sloan Management Review 39 (1) (1997) 9.

[46] B. Tessem, Individual empowerment of agile and non-agile software developers in small teams, Information and software technology 56 (8) (2014) 873–889.

[47] C. Wambui, A. Njuguna, The effect of financial governance on financial management system effectiveness in health oriented civil society organizations, American Journal of Health, Medicine and Nursing Practice 1 (1) (2016) 52–67.

[48] E. Defere, M. Abawa, K. Fenta, Prevalence and associated factors of internalized stigma among patients with severe mental disorder: The case of amanuel specialized mental health hospital, Ethiopian Renaissance Journal of Social Sciences and the Humanities 4 (2).

[49] T. Systa, P. Yu, H. Muller, Analyzing java software by combining metrics and program visualization, in: Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European, IEEE, 2000, pp. 199–208.

[50] H. M. Olague, L. H. Etzkorn, S. Gholston, S. Quattlebaum, Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes, IEEE Transactions on software Engineering 33 (6) (2007) 402–419.

[51] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software engineering 31 (10) (2005) 897–910.

[52] V. R. Basili, L. C. Briand, W. L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on software engineering 22 (10) (1996) 751–761.

[53] B. M. Goel, P. K. Bhatia, Analysis of reusability of object-oriented system using ck metrics, Analysis 60 (10).

[54] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al., Managing technical debt in software-reliant systems, in: Proceedings of the FSE/SDP workshop on Future of software engineering research, ACM, 2010, pp. 47–52.

[55] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, Ieee software 29 (6) (2012) 18–21.

[56] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on software engineering 26 (8) (2000) 786–796.

[57] E. Nasseri, S. Counsell, M. Shepperd, Class movement and re-location: An empirical study of java inheritance evolution, Journal of Systems and Software 83 (2) (2010) 303–315.

[58] J. W. Daly, A. Brooks, J. Miller, M. Roper, M. Wood, Evaluating inheritance depth on the maintainability of object-oriented software, Empirical Software Engineering 1 (2) (1996) 109–132.

[59] R. Harrison, S. Counsell, R. V. Nithi, Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems, Journal of Systems and Software 52 (2-3) (2000) 173–179.

[60] M. M. Lehman, On understanding laws, evolution, and conservation in the large-program life cycle, J. Syst. Softw. 1 (1984) 213–221.

[61] F. P. Brooks, Jr., The Mythical Man-month (Anniversary Ed.), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[62] W. G. Axinn, T. E. Fricke, A. Thornton, The microdemographic community-study approach: Improving survey data by integrating the ethnographic method, Sociological Methods & Research 20 (2) (1991) 187–217.

[63] A. Capiluppi, N. Ajienka, The relevance of application domains in empirical findings, in: Proceedings of the 2nd International Workshop on Software Health, IEEE Press, 2019, pp. 17–24.

[64] S. K. Shah, Motivation, governance, and the viability of hybrid forms in open source software development, Management Science 52 (7) (2006) 1000–1014.

[65] E. Capra, C. Francalanci, F. Merlo, An empirical study on the relationship between software design quality, development effort and governance in Open Source Projects, Software Engineering, IEEE Transactions on 34 (6) (2008) 765–782.

[66] Y.-T. Ma, K.-Q. He, B. Li, J. Liu, X.-Y. Zhou, A hybrid set of complexity metrics for large-scale object-oriented software systems, Journal of Computer Science and Technology 25 (6) (2010) 1184–1201.

[67] S. Counsell, S. Swift, J. Crampton, The interpretation and utility of three cohesion metrics for object-oriented design, ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (2) (2006) 123–149.

[68] Y. Zhou, H. Leung, Empirical analysis of object-oriented design metrics for predicting high and low severity faults, IEEE Transactions on software engineering 32 (10) (2006) 771–789.

[69] L. C. Briand, J. Wüst, J. W. Daly, D. V. Porter, Exploring the relationships between design measures and software quality in object-oriented systems, Journal of systems and software 51 (3) (2000) 245–273.

[70] M. Genero, M. Piattini, C. Calero, A survey of metrics for uml class diagrams, Journal of object technology 4 (9) (2005) 59–92.

[71] J. A. McQuillan, J. F. Power, On the application of software metrics to uml models, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2006, pp. 217–226.

[72] L. C. Briand, S. Morasca, V. R. Basili, Property-based software engineering measurement, IEEE transactions on software Engineering (1) (1996) 68–86.

[73] A. Bosu, M. Greiler, C. Bird, Characteristics of useful code reviews: An empirical study at microsoft, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 146–156.

[74] J. D. Herbsleb, D. Moitra, Global software development, IEEE software 18 (2) (2001) 16–20.

[75] J. D. Herbsleb, Global software engineering: The future of socio-technical coordination, in: Future of Software Engineering (FOSE'07), IEEE, 2007, pp. 188–198.

[76] A. Mockus, J. Herbsleb, Challenges of global software development, in: Proceedings seventh international software metrics symposium, IEEE, 2001, pp. 182–184.

[77] C. Parnin, C. Bird, E. Murphy-Hill, Java generics adoption: how new features are introduced, championed, or ignored, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, 2011, pp. 3–12.

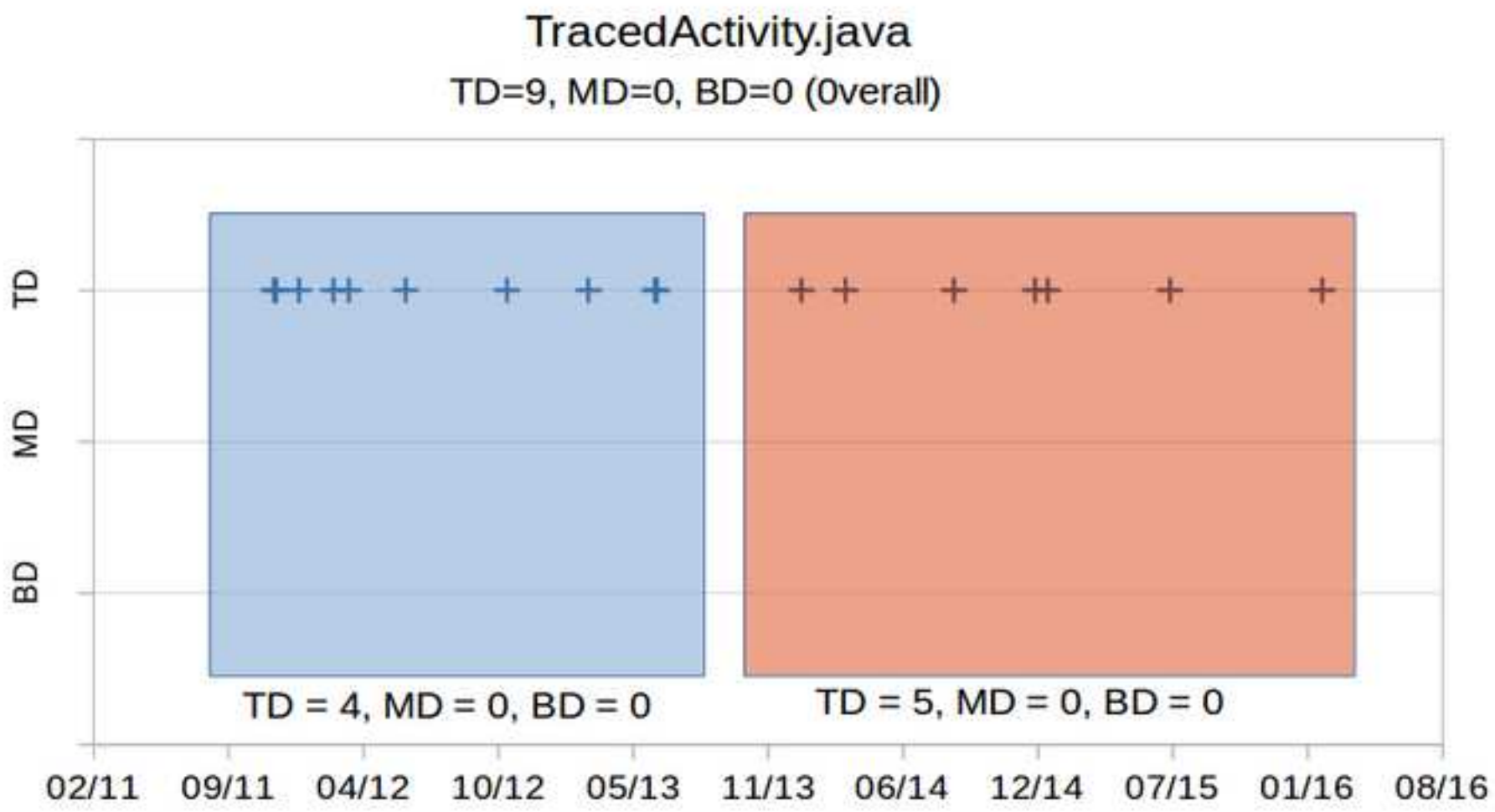
[78] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180.

***Biography**

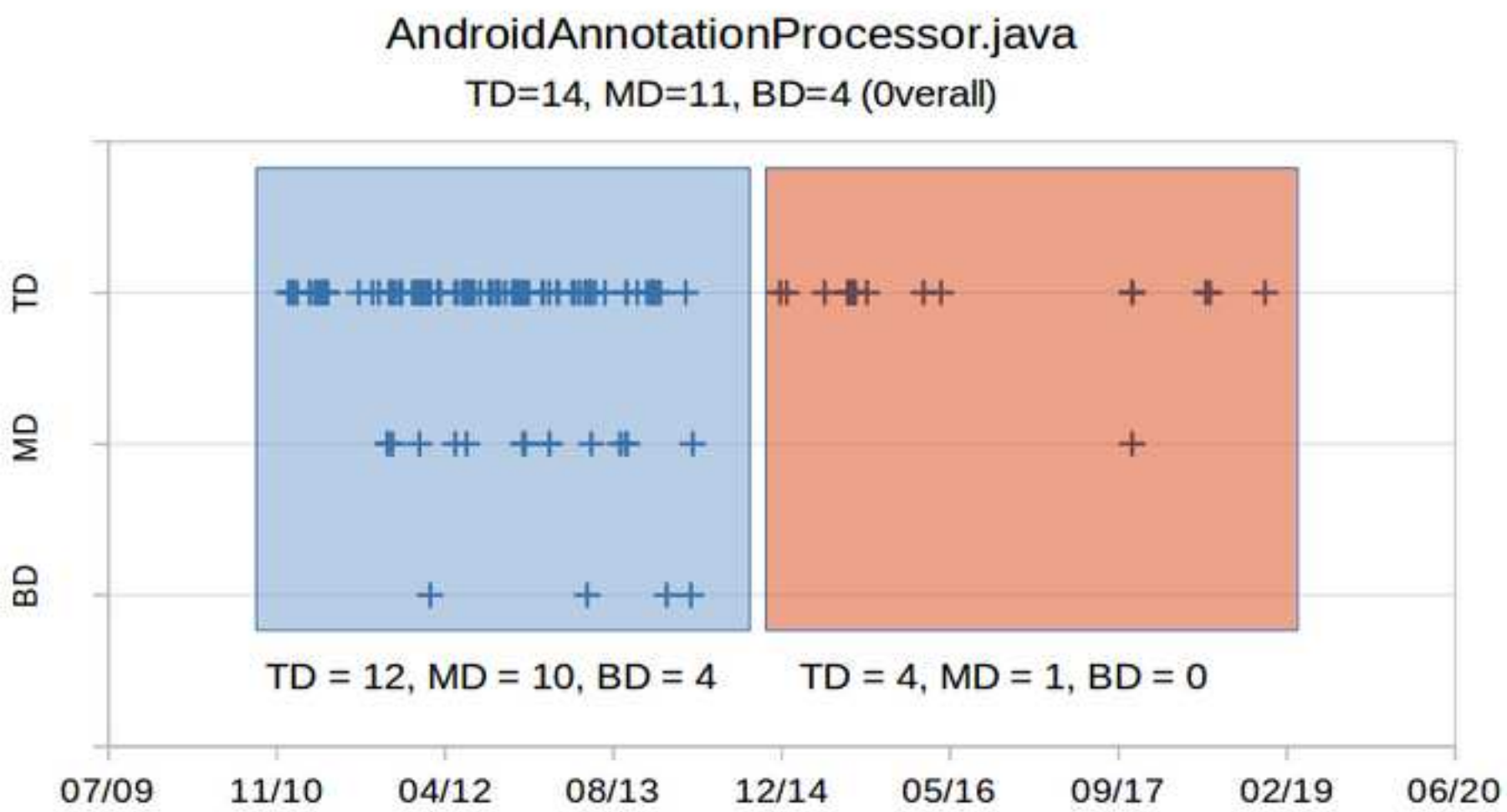
Andrea Capiluppi is a Senior Lecturer at Brunel University London;
Nemitari Ajienka is a Lecturer at Edge Hill University;
Steve Counsell is a Professor at Brunel University London

**Figure(s)**

Click here to download high resolution image



**Figure(s)**
Click here to download high resolution image



**LaTeX Source Files**

**Click here to download LaTeX Source Files: main.tex**

**LaTeX Source Files**

**Click here to download LaTeX Source Files: bibliography.bib**

***Credit Author Statement**

Andrea Capiluppi: Conceptualization, Methodology, Software, Writing - Original
draft preparation, Investigation, Data Curation. Steve Counsell: Conceptualization, Methodology, Investigation, Writing - Original draft preparation. Nemitari Ajienka: Data Curation, Investigation, Writing - Original Draft.

***Declaration of Interest Statement**

**Declaration of interests**

X 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.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: