# A Generic Methodology for Early Identification of Non-Maintainable Source Code Components through Analysis of Software Releases

Michail D. Papamichail*, Andreas L. Symeonidis

*Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Greece*
*mpapamic@issel.ee.auth.gr, asymeon@eng.auth.gr*

**Abstract**

**Context:** Contemporary development approaches consider that time-to-market is of utmost importance and assume that software projects are constantly evolving, driven by the continuously changing requirements of end-users. This practically requires an iterative process where software is changing by introducing new or updating existing software/user features, while at the same time continuing to support the stable ones. In order to ensure efficient software evolution, the need to produce maintainable software is evident.
**Objective:** In this work, we argue that non-maintainable software is not the outcome of a single change, but the consequence of a series of changes throughout the development lifecycle. To that end, we define a maintainability evaluation methodology across releases and employ various information residing in software repositories, so as to decide on the maintainability of software.
**Method:** Upon using the dropping of packages as a non-maintainability indicator (accompanied by a series of quality-related criteria), the proposed methodology involves using one-class-classification techniques for evaluating maintainability at a package level, on four different axes each targeting a primary source code property: complexity, cohesion, coupling, and inheritance.
**Results:** Given the qualitative and quantitative evaluation of our methodology, we argue that apart from providing accurate and interpretable maintainability evaluation at package level, we can also identify non-maintainable components at an early stage. This early stage is in many cases around 50% of the software package lifecycle.
**Conclusion:** Based on our findings, we conclude that modeling the trending behavior of certain static analysis metrics enables the effective identification of non-maintainable software components and thus can be a valuable tool for the software engineers.

*Keywords:* Software releases, maintainability evaluation, software quality, static analysis metrics, trend analysis

## 1. Introduction

Although highly efficient and with lots of merits, current software development approaches that suggest multiple sprints and contributors as the state-of-the-practice, while focusing on time-to-market and embrace the constant change of features, exhibit inherent weaknesses. This, usually, leaves little room for quality assessment and code optimization; it generates technical debt that is acknowledged and documented, however left for later engagement. Thus, the question regarding whether the produced software is maintainable arises.

Software maintainability is defined as the "degree of effectiveness and efficiency by which a product or system can be modified by the intended maintainers" [1]. The importance of maintainability as a software quality attribute is indicated by Robert L. Glass [2], who argues that maintaining software consumes about 40 to 80 percent of software costs. As a result, the need to provide methodologies and tools to evaluate and understand maintainability along with its interdependencies is evident.

Assessing maintainability is a non-trivial task considering that there are numerous and diverse criteria that actually influence the maintenance effort required for a software project. The need for setting a common ground towards understanding maintainability among the scientific community has led to the standardization of maintainability as a set of characteristics. Specifically, the product quality model introduced by the ISO/IEC 25010:2011 [1] describes maintainability as the composition of five characteristics: *Modularity*, *Reusability*, *Analyzability*, *Modifiability*, and *Testability*.

*Modularity* refers to the degree to which a piece of software is composed of discrete components, such that a change to one component has minimal impact on the others. *Reusability* reflects the degree to which a software component can be used in more than one systems, while *analyzability* measures the degree of effectiveness and efficiency with which it is possible to assess the impact of an intended change on a software project to one or more of its parts. *Modifiability* is closely related to analyzability and refers to how effectively a software project can be modified without introducing defects or degrading the existing quality. Finally, *testability* resembles the degree of effectiveness and efficiency with which test criteria can be established for a system and can be performed to determine whether those criteria have been met.

Various metrics have been proposed that can be used as the

information basis upon which maintainability predictors and evaluators can be built. Some of the proposed metrics may refer to the software development process [3, 4], while others focus on quantifying human resources [2, 5], examine the impact of code changes over the software development lifecycle [6, 7] or target the structure of the source code [8]. Employing metrics involves determining the appropriate thresholds, which is usually tackled by quality experts or by selecting predefined literature-based values. However, considering the fact that assessing maintainability is a multi-faceted and context-dependent problem, existing systems are usually restricted to certain use-case scenarios and provide subjective judgement.

In this work, we argue that non-maintainable software is not the outcome of a single change, but the consequence of a series of changes throughout the development process. Based on the imposed ground truth, we propose a generic maintainability evaluation methodology based on the analysis of software releases. We provide a proof-of-concept based on information residing in software repositories and analyze the extent to which the values of certain static analysis metrics influence the maintainability degree of software artifacts. The selected metrics correspond to primary source code properties that are closely related to the respective quality characteristics as proposed by ISO/IEC 25010:2011 [1]. Our system employs four maintainability evaluation models, each targeting a different source code property and thus is able to provide interpretable results.

The rest of this paper is organized as follows. Section 2 reviews current literature approaches on maintainability evaluation using static analysis metrics, while section 3 defines non-maintainablity and presents the concepts included in our analysis along with describing our benchmark dataset. Section 4 describes the designed methodology along with the training procedure of the proposed maintainability evaluation modes, while section 5 evaluates our methodology on a set of diverse axes. Finally, section 6 discusses threats to validity and section 7 provides insight for further research and concludes this paper.

## 2. Related Work

Assessing maintainability has drawn the attention of the scientific community for years and various preliminary approaches have been proposed that suggest the usage of metrics as a way to quantify maintainability. One of the most dominant approaches proposed the 'maintainability index' [9], which is a combination of the following three metrics: *Halstead Volume (HV)*, *Cyclomatic Complexity (CC)* and *Lines of Code (LOC)*. Various studies use maintainability index (MI) as is [9, 10, 11] or modify it [12] in order to best fit the empirically validated results. However, its applicability and accuracy are under discussion since it uses fixed thresholds which are both project and technology dependent [13].

In this context, deriving metrics thresholds and/or acceptable ranges is a non-trivial task and involves a series of parameters that need to be taken into consideration in order for the calculated values to be interpretable and conclusive. A common practice involves using predefined thresholds, usually defined by quality experts [14, 15]. However, such thresholds connot be generally applied and fail to incorporate the different characteristics of software projects. In an effort to overcome such restrictions, researchers have proposed deriving metrics thresholds by applying machine learning techniques on a benchmark repository. Ferreira et al. [16] propose a methodology for the estimation of metric thresholds by fitting the values of metrics into probability distributions, while Alves et. al. [17] follow a weight-based approach to derive thresholds by applying statistical analysis on the values of metrics. Other relevant approaches derive thresholds by employing bootstrapping [18] or ROC curve analysis [19]. Obviously, these approaches are subject to the type of projects selected to comprise the benchmark repository.

Instead of using the maintainability index (MI), certain approaches make use of various static analysis metrics in an effort to model the influence of different source code properties such as complexity, coupling, and cohesion on maintainability [20]. A representative example of such metrics are the widely-known CK metrics [8]. This approach also involves quality experts, who manually evaluate the usage of metrics and quantify their importance and applicability, usually by setting weights that correspond to the degree to which each metric influences a quality attribute [14]. However, expert-aided evaluation is limited only to certain scenarios due to time and resources constraints, especially when it comes to evaluating large and complex software projects.

Additional approaches that try to refrain from using experts often resort to applying machine learning techniques in an effort to model the relationships between metrics and maintainability. Koten and Gray [21] use empirical data so as to train a Bayesian Belief Network (BBN) for assessing software maintainability, while Cong and Liu [22] apply a Fuzzy C-Means clustering technique as the preprocessing step towards evaluating maintainability using a Support Vector Regression (SVR) model. Additional maintainability evaluation approaches suggest the usage of Artificial Neural Network (ANN) models [23] and Adaptive Multivariate Regression Splines (MARS) [24]. However, these approaches are still confined by the empirical evaluation of the software projects towards the formulation of the necessary ground truth which involves determining the software components that are non-maintainable.

A promising approach that seems to overcome the aforementioned limitation involves employing information residing in online software repositories and combining it with the values of static analysis metrics [25]. In this context, there are also some approaches that involve the analysis of software releases as a way for evaluating the maintainability degree of software. Samoladas et. al. [26] examine the maintainability degree of open-source software projects by investigating how the maintainability index (MI) changes over the releases. The projects selected in this study involve around fifteen (15) major releases. Another approach is the one of Fioravanti et. al. [27], where the authors analyze the impact of various static analysis metrics on the adaptive maintenance effort (AME) regarding a real system that has seven releases.

Although the maintainability evaluation approaches discus-

sed in the previous paragraphs can be effective for certain cases, their applicability in real-world scenarios is limited due to several reasons. At first, using predefined thresholds or certain formulas such as maintainability index leads to the creation of models unable to incorporate the various different characteristics of software projects. Automated evaluation approaches [23, 24, 25] seem to overcome these issues, but are still confined by the ground truth knowledge of quality experts who manually examine the source code in order to assess its degree of maintainability. Such approaches, apart from the fact that have high demands in terms of both time and resources, lead to subjective evaluation as each expert may perceive maintainability in a different manner. Additionally, another drawback of all the aforementioned approaches is that they are unable to predict non-maintainable software before occurrence. As a result, the evaluation cannot act preventively but identifies non-maintainable cases only when major refactoring is needed.

In this work, we build a generic and interpretable maintainability evaluation framework at a package level, which decomposes the maintainability degree of software into the quantified behavior of four primary source code properties: *complexity*, *cohesion*, *coupling*, and *inheritance*. The behavior of each property is modeled through the analysis of the values of certain static analysis metrics. In an effort to refrain from using fixed thresholds and to create models with predictive abilities, we have analyzed the progressing behavior of each metric and used their linear trend as a key modeling feature. As a result, we evaluate maintainability by taking into consideration the full development lifecycle of a software project. Furthermore, instead of using quality experts in order to manually evaluate the source code, we take advantage of information residing in software repositories and develop a methodology towards identifying non-maintainable software at a package level. Finally, we use the trends of the static analysis metrics of those packages in order to train four Support Vector Machines (SVMs) one-class classifiers, each targeting a specific source code property. Our models are able to effectively evaluate and predict at an earlier stage the maintainability degree of a given software project and they provide interpretable results regarding the properties that appear to have a problematic behavior.

## 3. Towards Modelling Maintainability

In this section, we discuss the basic concepts that constitute our maintainability evaluation methodology, which is built upon a full scale source code analysis of software projects. In specific, we define non-maintainability and present our releases analysis along with describing the formulated dataset that shall be used to train our maintainability evaluation models (more on the generated models in Section 4).

### *3.1. Defining non-maintainability*

As already noted, given the ISO/IEC 25010:2011 standard [1], maintainability comprises the following sub-characteristics: *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability*. In this work, we evaluate maintainability using the progressing behaviour of the values of various static analysis metrics that quantify the four source code properties related to the aforementioned maintainability characteristics: *complexity*, *coupling*, *inheritance*, and *cohesion*.

Table 1 presents the relation of the selected source code properties with the maintainability-related quality characteristics. The "✓" suggests that a given characteristic is influenced by the respective source code property, while the arrow (↑ or ↓) inside the parenthesis defines whether this influence is positive (↑) or negative (↓). Positive influence denotes that the greater the presence of the source code property (as quantified by the values of the respective static analysis metrics), the stronger the characteristic (and thus the maintainability degree of the component under evaluation). On the other hand, negative impact denotes that the absence of the respective property leads to a more maintainable component. As shown in the table, coupling, inheritance and cohesion are associated with all five maintainability-related characteristics while complexity appears to influence three of them. In addition, complexity, coupling and cohesion appear to have a negative impact on the maintainability degree of software components, while on the other hand the higher level of cohesion lead to more maintainable components. At this point it is worth mentioning that as far as inheritance is concerned, there are also studies that suggest that inheritance may also increase the degree to which a software component is maintainable from a modifiability perspective [28]. However, this increase is still based on the absolute values of certain inheritance metrics, which need to lay between certain intervals that are acceptable from a quality perspective.

Table 1: Relation of Maintainability with Source Code Properties [15, 29, 30]

| | **Source Code Properties** | | | |
|---|---|---|---|---|
| **Maintainability Characteristics** | **Complexity** | **Coupling** | **Inheritance** | **Cohesion** |
| Modularity | — | ✓(↓) | ✓(↓) | ✓(↑) |
| Reusability | — | ✓(↓) | ✓(↓) | ✓(↑) |
| Analyzability | ✓(↓) | ✓(↓) | ✓(↓) | ✓(↑) |
| Modifiability | ✓(↓) | ✓(↓) | ✓(↓) | ✓(↑) |
| Testability | ✓(↓) | ✓(↓) | ✓(↓) | ✓(↑) |

↑ : Positive influence.
↓ : Negative influence.

In the context of this work, we employ information residing in software repositories (e.g. GitHub) and combine it with the progressing behavior of metrics that quantify the aforementioned maintainability-related source code properties. We construct a methodology for evaluating the maintainability degree of software projects and argue that dropping software packages (removing them from the codebase), if combined with certain quality-related criteria, can be used as a non-maintainability indicator. Towards this direction, we analyze the lifecycle of packages that have been dropped from certain software projects (considered as candidates for non-maintainability occurrence) as reflected in the progressing behavior of a series of static

analysis metrics. This analysis involves the application of a series of quality-related criteria that indicate which candidates are actually non-maintainable along with locating the property (or properties) that cause their being non-maintainable.

### 3.2. Metrics Behavior Extraction

In an effort to build a generic and reliable maintainability evaluation system, we resort in analyzing every release of the software project since the beginning of its development process. Such an exhaustive analysis results into the computation of a large set of static analysis metrics for various timeframes and at the same time introduces their progressing behaviour as a key modeling feature. Our design choice to monitor the progressing behaviour of metrics by using releases as our reference point (instead of i.e. commits or week index) relies in the fact that releases resemble the actual maturity level of a certain piece of software at a given timeframe, as they incorporate all the design choices made by the development team both from architectural and functional perspective.

Figure 1 illustrates the progressing behaviour of the Weighted Methods per Class (WMC) metric over the lifecycle of a specific package included in the *elasticsearch*[1] project. The lifecycle of the package refers to the time period starting from the first release it existed in the project until the release it was dropped. In this case, the lifecycle of the package under examination includes 50 releases starting at release 0.90.13 up to the release 2.1.2. As for the progressing behaviour, the solid line refers to the absolute values of the WMC metric, while the dashed line refers to the calculated linear trend. Our methodology considers the slope of the linear trend of each metric computed for every package within the project as the key feature for determining the maintainability level of the package.

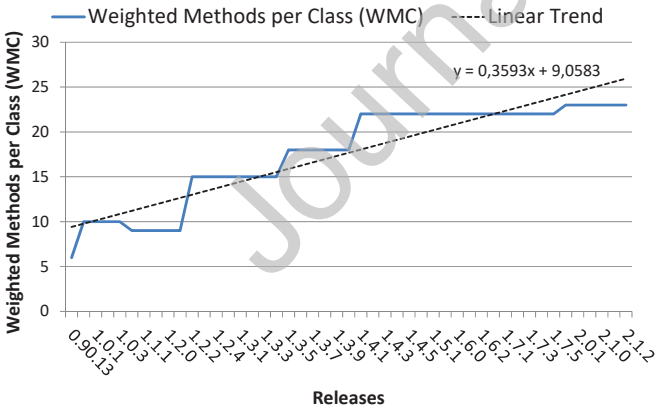



Figure 1: Output of releases analysis of WMC metric for a certain package.

### 3.3. Dataset

Our dataset consists of a large set of static analysis metrics calculated for every class and every package included in the 127 releases of the project *elasticsearch* starting from the release 0.4.0 up to the release 2.2.0. Full reference of the computed metrics along with their computation level (method or class) can be found in Table 3 and online in [31]. Given that all static analysis metrics are computed at class level, we generate the value of each metric at package level as the average of the values regarding all classes included in the package. For the computation of the aforementioned static analysis metrics, the tool *Sourcemeter*[2] was used. The selection of the elasticsearch project (made using the GitHub API[3]) for our benchmark dataset relies on the fact that it reflects the characteristics of a long and complex project . It has a very long lifecycle (more than 7 years) and a large number of contributors (>800) who change on a regular basis. In addition, considering the large amount of releases (127) and open issues (>1,100), it is obvious that it involves the continuous introduction of new features, while at the same time maintaining the pre-existing ones.

Table 2: Dataset Statistics

| **Statistics regarding the constructed dataset** | |
|---|---|
| Number of releases | 127 |
| Total Number of Methods | 3,535,960 |
| Total Number of Classes | 618,157 |
| Total Number of Packages | 81,831 |
| Total lines of code (LOC) | 56,333,352 |

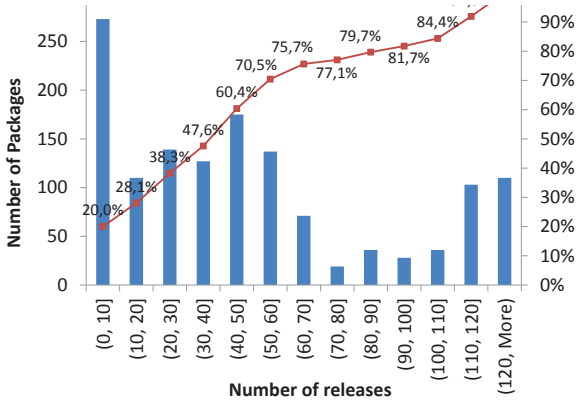



Figure 2: Histogram of the number of releases included in the lifecycle of all packages of the elasticsearch project.

Furthermore, towards modeling maintainability in a generic manner, analysis at a package level is performed. Elasticsearch contains a large number of different packages (>1,500) that differ both in size and in the number of releases (length of lifecycle). Figure 2 presents the histogram regarding the length of lifecycle (as reflected in the number of releases) for all packages included in the elasticsearch project. The bins (x-axis) refer to the number of releases, while primary y-axis refers to the number of packages. The number of releases included in the lifecycle of every package is computed by monitoring the first and the least release the package was present in the project. The

[1] https://github.com/elastic/elasticsearch

[2] https://www.sourcemeter.com/

[3] https://developer.github.com/v3/search

Table 3: Overview of the computed Static Analysis Metrics [31]

| Static Analysis Metrics | | | Computation Levels | |
|---|---|---|---|---|
| **Property** | **Name** | **Description** | **Method** | **Class** |
| ***Complexity*** | NL | Nesting Level | X | X |
| | NLE | Nesting Level Else-If | X | X |
| | WMC | Weighted Methods per Class | — | X |
| ***Coupling*** | CBO | Coupling Between Object classes | — | X |
| | CBOI | Coupling Between Object classes Inverse | — | X |
| | NII | Number of Incoming Invocations | X | X |
| | NOI | Number of Outgoing Invocations | X | X |
| | RFC | Response set For Class | — | X |
| ***Cohesion*** | LCOM5 | Lack of Cohesion in Methods 5 | — | X |
| ***Inheritance*** | DIT | Depth of Inheritance Tree | — | X |
| | NOA | Number of Ancestors | — | X |
| | NOC | Number of Children | — | X |
| | NOD | Number of Descendants | — | X |
| | NOP | Number of Parents | — | X |

solid line depicts the cumulative percentage (secondary y-axis). For instance, as shown in the figure, 20% of the packages included in the elasticsearch project appear to have a lifecycle with length that lies in the interval (0, 10].

Additionally, Figure 3 presents the distribution of the average lines of code (LOC) per class (Figure 3(a)) and the number of classes (NCL) (Figure 3(b)) for every package of the last analyzed version of elasticsearch (version 2.2.0). Given the large range of the values of both metrics, the scale of y axis is logarithmic in both figures. Quartiles for each metric are depicted in the graph. We argue that such a project with packages of varying duration and size provide a good benchmark set with many different scenarios, allowing our maintainability models to generalize.

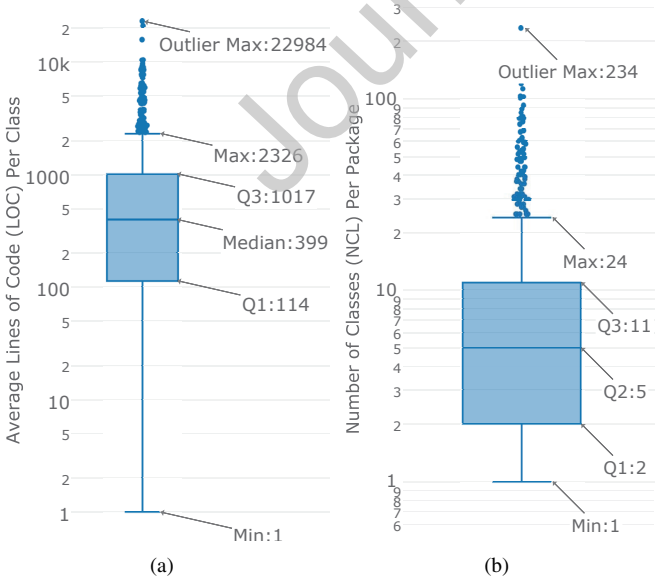



Figure 3: Boxplots of 3(a) Average Lines of Code per Class and 3(b) of Number of Classes for all packages of elasticsearch release 2.2.0.

Our methodology can be applied at any level of granularity (project, package, class, method). However, our design choice to analyze at package level originates from the intention to accurately model the behavior of metrics through trend analysis. Specifically, in order to examine the optimal level, a comparative analysis of the predictive ability of the computed trends at package and class level was performed. The criteria used to evaluate this ability were the coefficient of determination ($R^2$) and the root mean square error (RMSE) of the metrics. The calculation formulas regarding the aforementioned evaluation metrics are given by the following equations:

$$R^2 = 1 - \frac{\sum_{i=0}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{N}(y_i - \bar{y}_i)^2} \tag{1}$$

$$RMSE = \sqrt{\frac{\sum_{i=0}^{N}(y_i - \hat{y}_i)^2}{N}} \tag{2}$$

Equation 1 refers to the coefficient of determination which corresponds to the proportion of the variance in the dependent variable that is predictable from the independent variable, while equation 2 refers to the root mean square error which represents the sample standard deviation of the differences between the predicted values and the observed values. In our case, the independent variable corresponds to the release number, while the dependent variable corresponds to the respective static analysis metric. In both equations, $y_i$ refers to the actual value, $\hat{y}_i$ to the predicted value, while N denotes the number of samples.

Figure 5 illustrates a representative case regarding the evaluation of modeling the behaviour of a given metric at both class and package levels. In specific, we calculated the linear trends for a specific package along with each one of its classes. The figure depicts the values for both the coefficient of determination (bars) and the RMSE (line). The first ten instances refer

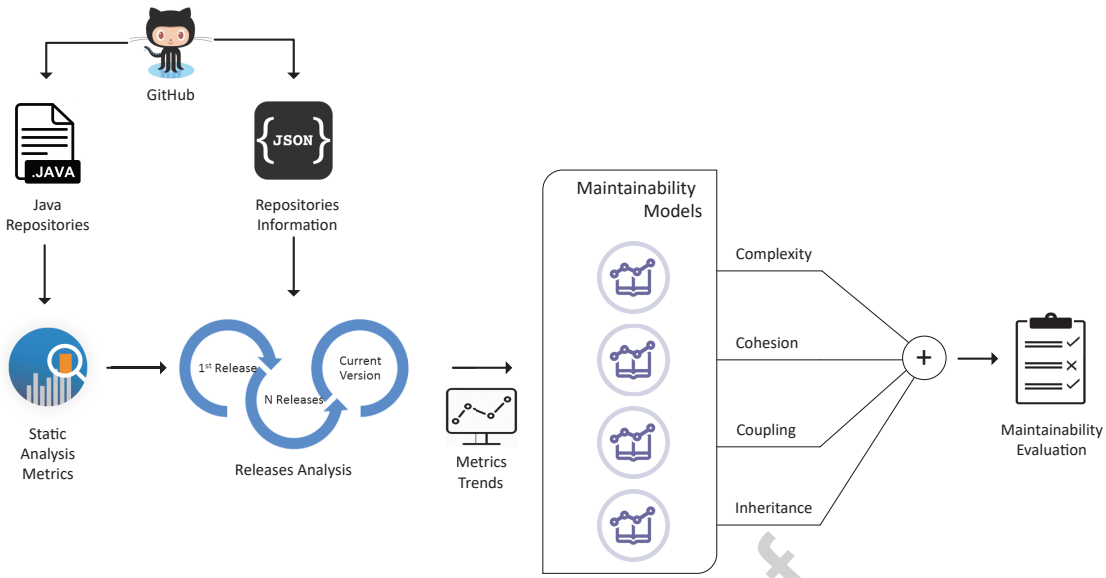



Figure 4: System Overview

to the classes (blue bars), while the last one (red bar) to the package they belong. The first four classes have the same values for the given metric throughout their lifecycle and thus their behaviour contains no valuable information. Comparing the results between the rest six classes and the package, it is obvious that trend analysis at package level provides the best combination of RMSE and $R^2$. From a metrics perspective the aforementioned results are expected since the fluctuation in the values of the static analysis metrics at class level is much higher as compared to the case of packages (the change proneness across releases at class level is much higher than at package level). As a result, modeling packages behaviour can provide more accurate results.

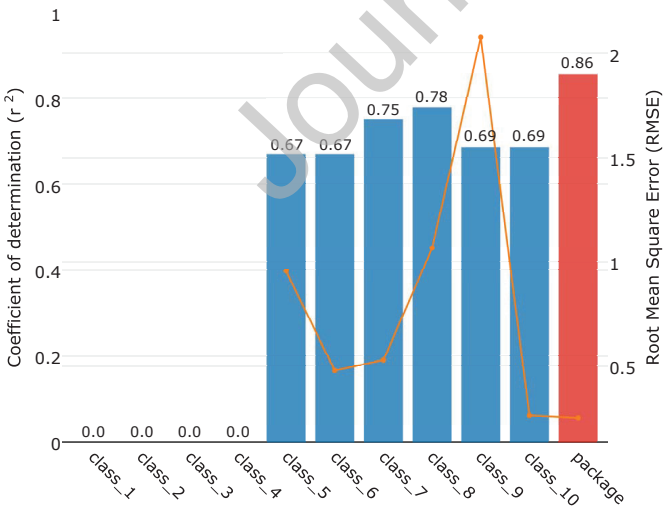



Figure 5: Comparative evaluation regarding modelling the behaviour of a given metric at class vs at package level.

## 4. The Designed System

In this section, we present our analysis towards the construction of our maintainability evaluation models, each targeting a specific source code property.

### *4.1. Overview*

Figure 4 provides a general overview of the designed maintainability evaluation system, while Figure 6 illustrates the steps of the proposed methodology for assessing the maintainability degree of software components. Our methodology comprises five distinct steps. At first, we retrieve the information regarding the releases of the project upon which our maintainability evaluation models will be built (step a). We use this information in order to calculate the lifecycle of every package included in the project. The term lifecycle refers to the time period between the first and the last release the package existed in the software project. Having retrieved the information regarding the releases of the software project along with the source code of each release, the next step (step b) involves performing static analysis in order to compute the values of various static analysis metrics that refer to four primary source code properties: *complexity*, *cohesion*, *coupling*, and *inheritance*. These source code properties constitute the axes upon which we assess the maintainability degree of software components. At this point, it is worth noting that given the fact that the static analysis metrics are computed at class level, this step involves aggregating the computed values at package level. During this process, in order to compute the aggregated values for a certain package, we use only its child classes, while averaging has been selected as the aggregation formula.

After having computed the aggregated values (one aggregated value for each release) of all metrics for every package included in the software project, the next step (step c) involves using them in order to calculate the trend of each metric, which reflects its progressing behaviour. We calculate one trend for each

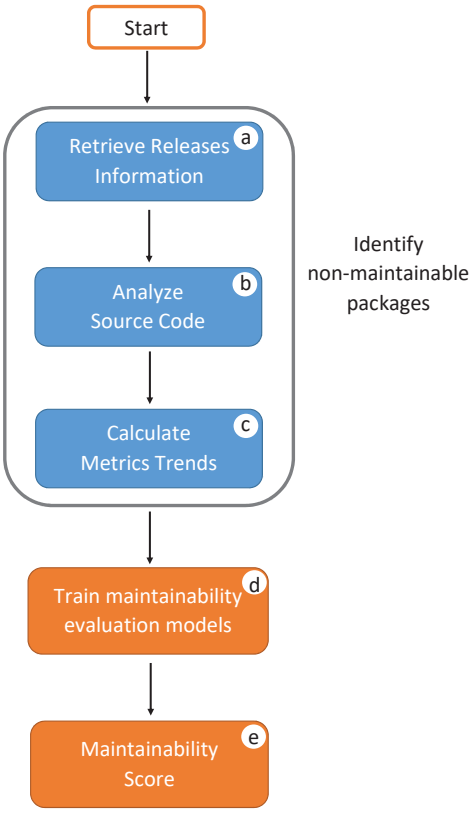



Figure 6: Methodology Flowchart

metric and each package. Using these trends along with the lifecycle information regarding all packages enables us to identify the packages that are considered as non-maintainable and thus the ones that constitute our ground truth towards the construction of our maintainability evaluation models. The fourth step (step d) involves using the formulated ground truth in order to train four maintainability evaluation models, each targeting a different source code property. The training process for each model involves applying one class classification using Support Vector Machines. The design choice for training four models (instead of one) relies in the fact that our primary design principle was to provide interpretable results that can lead to certain actionable recommendations towards improving the maintainability degree of the software project under evaluation. Finally, the last step (step e) involves combining the output of all four models into a single score that reflects the maintainability degree of the package under evaluation.

### 4.2. Linear Trend Calculation

As already noted, the lifecycle of every package refers to the time which starts at the release when the package was created and ends at the last release the package existed in the project. This last release could either be the latest release, which means that the package is still present in the software project or it can be a past release. which denotes that the package has been dropped. Thus, the information regarding the lifecycle regarding a given package can be considered as a list of time-series containing the values of all computed for all releases it existed in the project. The values of each metric are used to compute its linear trend based on the following formulas:

$$Y_i = \alpha \cdot X_i + \beta \tag{3}$$

$$\sum_{i=0}^{N} [(\alpha \cdot X_i + \beta) - Y_i]^2 \tag{4}$$

Equation 3 refers to the representation of the linear trend, where parameter $a$ refers to the slope and parameter $b$ to a constant factor. These two parameters are calculated so as to minimize the sum of squared deviations from the trend line, calculated through (4).

### 4.3. Preprocessing

The preprocessing step involves using the computed linear trends in order to determine the non-maintainable packages along with the respective code property (or properties) that cause the non-maintainability. The first step involves using the repository information (releases and source code) in order to locate all packages that have been dropped. This identification process uses the names of all packages along with their dependencies information contained in each release. These packages constitute the candidates for non-maintainable packages.

Considering all dropped packages as non-maintainable is not effective as a package drop may also imply a change of functionality or a design choice that has nothing to do with non-maintainability. To that end, we apply a series of filters towards eliminating biases that may occur by false-positives. The first step involves keeping only the packages that are present for more than 3 releases and are then being dropped. The minimum of three releases of presence was used in an effort to avoid cases where the introduced functionality was dropped due to an instant design choice (avoid one-off components). In addition, in an effort to refrain from biases that occur from packages that are being dropped based on obsolete functionality and/or 3rd party decisions that involve no quality criteria, we only consider as non-maintainable the packages for which at least one metric has a trend that negatively affects their maintainability degree. Finally, in an effort to further eliminate false-positives, we apply code clone detection techniques in order to identify the cases when two packages are identical and thus exclude them from our analysis. This additional filter enables us to accurately identify the cases of renamed packages, which could have been identified as dropped.

After having selected the non-maintainable package candidates, the next step involves the examination of the trend of each metric for every non-maintainable package in order to decide for the property (or properties) that is (or are) responsible for its being non-maintainable. For instance, dropped packages where the complexity-related metrics (WMC, NL and NLE) exhibit high positive trends are considered non-maintainable and the *complexity* property is flagged as responsible. Upon applying the same methodology for all properties under evaluation: *complexity*, *cohesion*, *coupling* and *inheritance*, we are able to locate non-maintainability occurrences and thus create the necessary ground truth in order to predict them.

### 4.4. Models Construction

After having flagged the properties that are responsible for the non-maintainable packages, next comes the model construction phase which involves the training of four one-class classifiers using support vector machines (SVMs). Each one-class classifier is used in order to evaluate the maintainability degree of a given package based on a specific source code property. The selection of four models instead of one originates from the fact that our primary target was building a configurable and interpretable maintainability evaluation system able to adapt to the developers' needs.

Using one-class classification originates from the fact that based on our ground truth information which originate from code repositories, we are unable to come to a safe conclusion on whether a package is maintainable but only identify the ones that are non-maintainable. Consequently, our dataset includes only non-maintainable packages. For training each model, we use only the non-maintainable packages flagged for the respective source code property. Table 4 presents the number of packages identified as non-maintainable for each source code property, while Table 5 provides information regarding the meta-parameters' selection process for each one-class classifier. The selection process is based on the percentage of False-Negatives (FN) and optimizes the values of three meta-parameters: ***nu***, which corresponds to the fraction of training errors and a lower bound of the fraction of support vectors, ***gamma***, which is the kernel coefficient that reflects how far the influence of a single training example reaches, and ***cost***, which trades off misclassification of training examples against simplicity of the decision surface. As shown in Table 4, coupling and complexity are the dominant properties responsible for most non-maintainable occurrences.

Table 4: Number of Non-Maintainable Packages per Source Code Property

| Source Code Property | # non-maintainable packages |
|---|---|
| Complexity | 139 |
| Cohesion | 33 |
| Coupling | 246 |
| Inheritance | 72 |

Table 5: Statistics regarding the selection of Meta-Parameters for the Constructed Models based on the Percentage of False-Negatives (FN)

| Maintainability | Training Meta-parameters | | | |
|---|---|---|---|---|
| Evaluation Model | Nu | Gamma | Cost | FN (%) |
| Complexity | 0.01 | 0.15 | 512 | 0.72% |
| Cohesion | 0.03 | 0.05 | 512 | 3.03% |
| Coupling | 0.03 | 0.06 | 256 | 2.78% |
| Inheritance | 0.03 | 0.1 | 64 | 2.03% |

The following paragraphs present the training results regrading the trained maintainability evaluation models, each targeting a different primary source code property.

- **Complexity Model**

The dataset includes the trends regarding three static analysis metrics that are related to complexity: NL, NLE, and WMC. As shown in Table 5, the selected values for the nu, gamma and cost parameters are 0.01 and 0.15 and 512, respectively. The percentage of the false negatives (FN) is 0.719%.

- **Cohesion Model**

The dataset includes the trends regarding the LCOM5 metric, which corresponds to the number of coherent classes each class could be split. In a similar manner to the aforementioned analysis, the selected values are 0.03, 0.05 and 512 for the nu, gamma and cost parameters, respectively, while the percentage of the false negatives is 3.03%.

- **Coupling Model**

The dataset includes the trends regarding five static analysis metrics that are related to coupling: CBO, CBOI, NII, NOI, and RFC. For the coupling model, the selected values are 0.03, 0.06 and 256 for the nu, gamma and cost parameters, respectively, while the percentage of the false negatives is 2.778%.

- **Inheritance Model**

The dataset includes the trends regarding five static analysis metrics that are related to inheritance: DIT, NOA, NOC, NOD, and NOP. For the inheritance model, the selected values are 0.03, 0.1 and 64 for the nu, gamma and cost parameters, respectively, while the percentage of the false negatives is 2.033%.

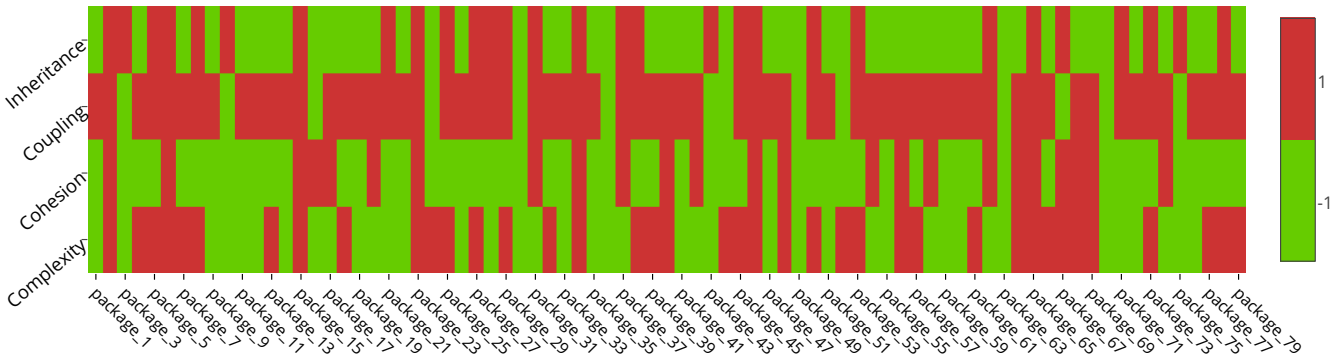
## 5. Proposed Methodology Evaluation

The evaluation of our maintainability evaluation methodology is performed around three axes: a) the system's ability to evaluate the maintainability degree of the packages included in a number of randomly selected diverse projects along with its ability to provide interpretable results, b) the effectiveness of our maintainability evaluation models to locate and predict non-maintainability before occurrence (early prediction) and c) the effectiveness of our methodology against the widely used maintainability index (MI).

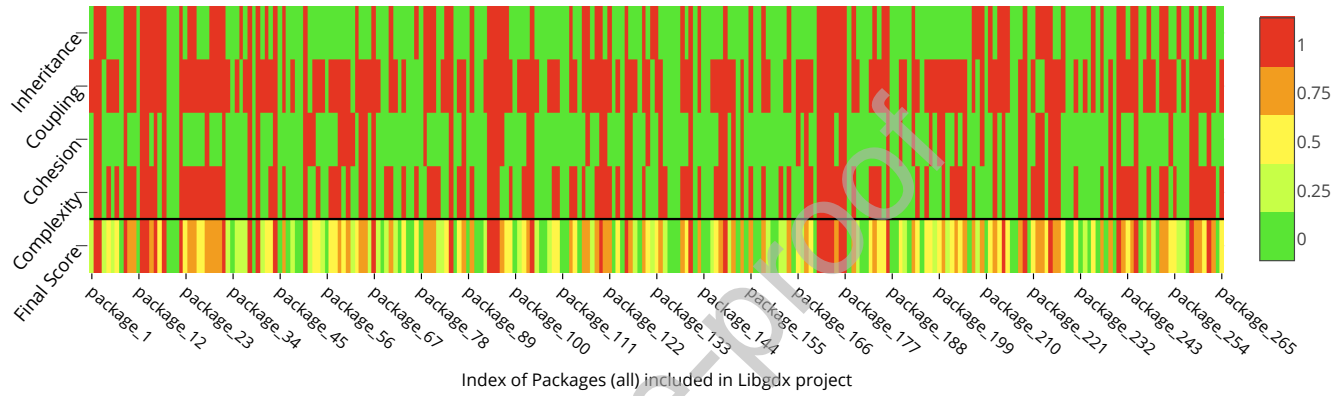
### 5.1. Evaluating the appropriateness of the benchmark dataset

At first, we evaluated the ability of our system to identify non-maintainable software at package level, and thus the appropriateness of our benchmark dataset, by applying our maintainability evaluation models on four independent randomly selected GitHub repositories. Namely, the selected repositories are libgdx[4], Apache Struts[5], Hibernate[6] and Spring Framework[7]. Similar to case of the selection of the benchmark repository,

[4] https://github.com/libgdx/libgdx
[5] https://github.com/apache/struts
[6] https://github.com/hibernate/hibernate-orm
[7] https://github.com/spring-projects/spring-framework




(a)




(b)

Figure 7: Maintainability evaluation results for 7(a) the non-maintainable packages and 7(b) for every package included in libdgx project

the selection was made using the GitHub API. Certain statistics regarding the aforementioned repositories are presented in Table 6. Given the number of releases of the repositories used for the evaluation of our methodology, which varies from 35 (Libgdx project) to 156 (Hibernate project), along with their differences in size, we argue that these projects cover a wide range of diverse scenarios and thus enable us to evaluate the ability of our maintainability evaluation models to generalize. In order to evaluate our models, we performed static analysis in all releases of the aforementioned projects; practically, we analyzed more than 138 millions lines of code, in order to validate our methodology.

Table 6: Statistics of Evaluation Repositories

| **Metric** | **Libgdx** | **Apache Struts** | **Hibernate** | **Spring Framework** |
|---|---|---|---|---|
| Releases | 35 | 108 | 156 | 120 |
| Methods | 1.2M | 1.5M | 8.2M | 6,1M |
| Classes | 101K | 225.8K | 1.2M | 1,14M |
| Packages | 5.6K | 20.3K | 164K | 55,4K |
| LOC | 11.1M | 12.9M | 66.4M | 47,9M |

At this point, we will provide a detailed description of our evaluation approach along with a graphical representation of our maintainability evaluation results using libgdx project as our reference repository. The full summary of the evaluation results regarding all four repositories are presented in Tables 7 and 8. Libgdx project has 35 releases, which contain more than 11 million lines of code. Throughout its development process, the project appears to have 265 unique packages out of which 79 are identified as non-maintainable following the procedure described in the previous sections. Figure 7(a) illustrates the maintainability evaluation results regarding the packages of libgdx project that are identified as non-maintainable. The figure presents the results regarding all models each evaluating a different source code property. Each row of the heatmap illustrates the maintainability evaluation results based on a different property. Green color denotes that the package is considered as maintainable regarding the respective code property, while red indicates that the package is considered as non-maintainable. We also evaluated the sensitivity of our maintainability evaluation models (given in Table 7), which in the case of Libgdx project is 93.67%, as our models were able to correctly identify the 74 out of the 79 non-maintainable packages. It is worth noticing that 60.75% (48 out of 79) of the non-maintainable packages for the libgdx project were found non-maintainable by more than one models, which indicates problematic behav-

ior in more than one code properties. This is expected as code changes that affect a specific property often influence other properties, too.

Figure 7(b) depicts the output of our maintainability evaluation system when assessing a full repository (both maintainable and non-maintainable packages). In specific, Figure 7(b) illustrates the maintainability evaluation results for every unique package (265 in total) included in the 35 releases of the libgdx project. Similar to the case of Figure 7(a), each one of the first four rows of the heatmap depicts the results based on a certain property, while the last row provides a maintainability risk value which represents the percentage of the evaluated source code properties that appear to lead to non-maintainable status. Zero (represented with green color) denotes that the package is maintainable, while one (represented with red color) denotes that the behavior of all code properties is found to be problematic. The available levels are 0, 0.25, 0.5, 0.75 and 1.

Table 7: Maintainability Evaluation Results

| Projects | Total Packages | Non-maintainable Packages | Sensitivity |
|---|---|---|---|
| Libgdx | 265 | 79 | 93.67% |
| Apache Struts | 260 | 55 | 72.72% |
| Hibernate | 1845 | 494 | 81.37% |
| Spring Framework | 684 | 127 | 83.46% |

Table 8: Percentage of packages evaluated as non-maintainable per Source Code Property

| Property | Libgdx | Apache Struts | Hibernate | Spring |
|---|---|---|---|---|
| Complexity | 49.36% | 43.63% | 54.25% | 9.44% |
| Cohesion | 30.37% | 12.72% | 31.78% | 23.62% |
| Coupling | 82.27% | 16.36% | 51.01% | 21.25% |
| Inheritance | 36.71% | 36.36% | 18.42% | 59.05% |

Tables 7 and 8 present the maintainability evaluation results regarding the four different repositories used for evaluation. Table 7 provides information regarding the ability of our maintainability evaluation models to identify non-maintainable packages (based on the sensitivity criterion), while Table 8 presents the results regarding the source code properties found responsible for the non-maintainable components. As shown in Table 7, the sensitivity (true-positive-rate) of our maintainability evaluation approach varies from 72.72% (apache struts) to 93.67% (libgdx project), which indicates that our models are able to effectively identify non-maintainable packages long before they become non-maintainable. The selection of sensitivity (True-Positive-Rate) for evaluating our approach relies in the fact that we can only come to a safe conclusion for non-maintainable packages as a package is flagged as non-maintainable under certain conditions (see subsection 4.3).

As for the results regarding the properties responsible for the non-maintainable packages (see Table 8), they indicate that while all four code properties appear to play a very important role in developing maintainable software, their degree of importance differs among repositories. For instance, in the case of libgdx project, complexity and coupling appear to be the dominant properties that result in the majority of non-maintainable packages, while in the case of the spring framework, inheritance appears to be the dominant property. This is totally expected as the field of application of every software project has a great impact on both functional and non-functional requirements and thus influences maintainability in a different manner.

Given the above, we argue that our maintainability evaluation system is able to effectively identify non-maintainable packages along with providing interpretable results regarding the code properties that appear to have problematic behaviour.

### *5.2. Non-maintainability prediction before occurrence*

Identifying non-maintainable software at a late stage is both time and resources demanding given that major refactoring is needed. This fact has also a strong impact on the development costs. Consequently, one of the primary targets of our maintainability evaluation methodology is its ability to effectively identify non-maintainable software at an earlier stage (before their becoming non-maintainable) and thus act in a preventive rather than in a corrective manner.

Towards this direction, we evaluate the ability of our system to provide early predictions. The early prediction refers to how many releases earlier (before occurrence) our models are able to correctly identify non-maintainability. To that end, we calculated the metrics trends for every package and for every release taking into account only the previous releases. For instance, for a given package that appears to be in the project for 20 releases and is then being dropped, we use only the values of the first 15 releases (as if our current timestamp was the 15th release) in order to calculate the metrics' trends and use our models to evaluate its maintainability degree. If we successfully identify the package as non-maintainable, then we have a correct prediction 5 releases ahead, which corresponds to the 25% of the package lifecycle. Using this strategy for all releases, we were able to assess the maintainability degree of each package for every release, as if it was the current release and thus calculate the number of releases ahead that our models provide correct evaluation. The number of releases was then transformed into the percentage of lifecycle for each package by dividing it with the total number of releases. As already noted, we use the term lifecycle for a package in order to refer to the time period between the first and the last release it existed in the software project.

Following a similar strategy to the previous subsection, we used libgdx project as our reference repository so as to demonstrate the percentage of lifecycle ahead of which our models

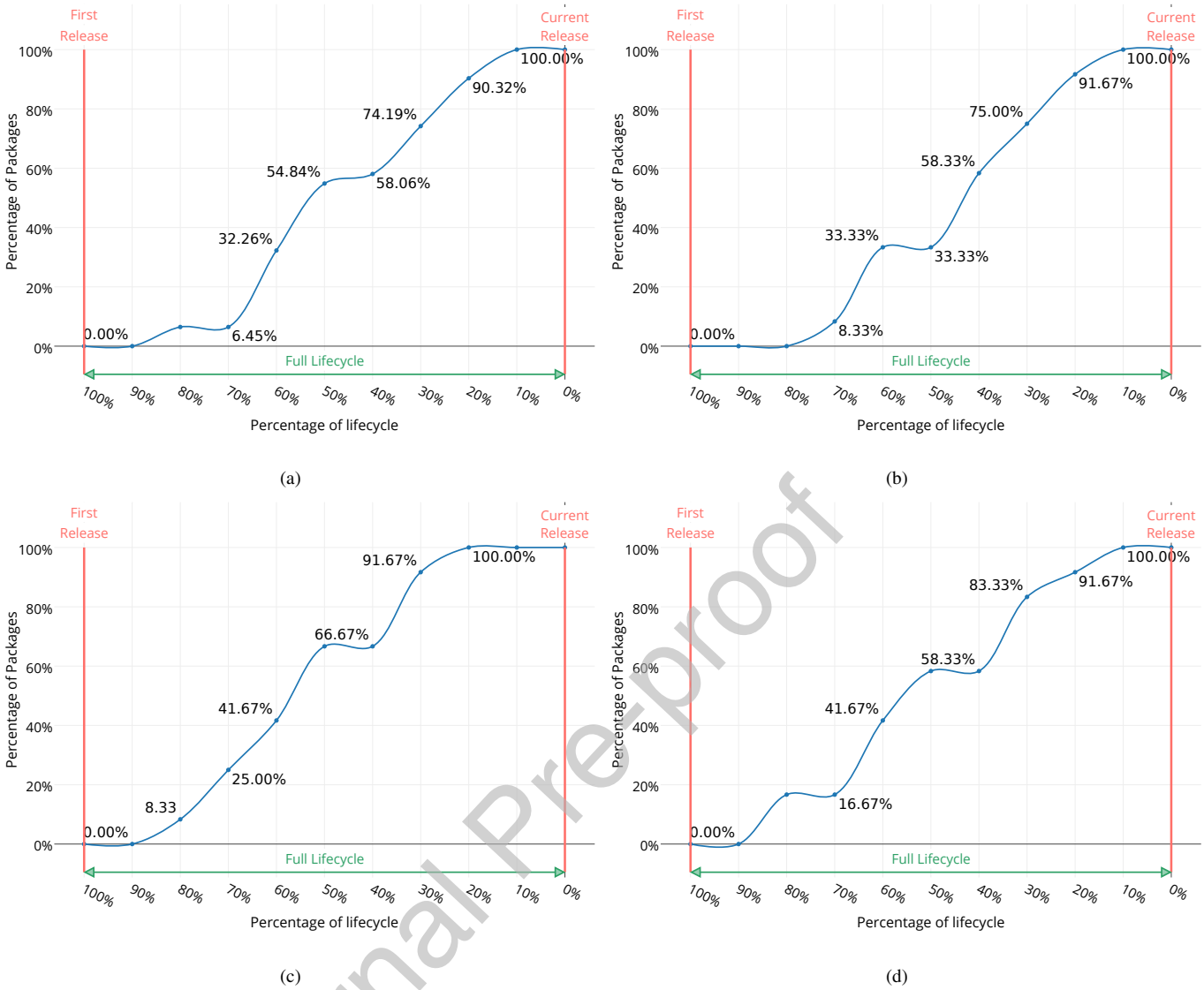



Figure 8: Histograms illustrating maintainability prediction results before occurrence for 8(a) the complexity model, 8(b) the coupling model, 8(c) the cohesion model and 8(d) the inheritance model

can provide accurate predictions (see Figure 8). The maintainability prediction results before occurrence for all four evaluation repositories are given in Table 9, while Figure 8 illustrates the ability of each model to provide early predictions of non-maintainability for the libgdx project. In specific, the y axis corresponds to the percentage of packages correctly identified as non-maintainable, while the x axis refers to the percentage of lifecycle starting from the current release (0% ahead) up to the beginning of the lifecycle (100% ahead). The blue line depicts how the accuracy of each maintainability evaluation model decreases when performing early predictions. Figures 8(a), 8(b), 8(c), and 8(d) refer to the complexity, coupling, cohesion, and inheritance model, respectively. The results indicate that all four models are able to provide correct evaluation (for almost 60% of the non-maintainable packages) at least 40% earlier. Table 9 presents the results regarding the average earlier prediction for each model (first four rows) along with the aggregated average correct early prediction using all four models (declared as total). In the case of the aggregated early prediction, we assume that a package is considered as non-maintainable if it is evaluated as non-maintainable by any of the models. As a result, it is expected to exhibit higher values than in the case of evaluating the early predictions of each model separately.

Table 9: Average Correct Early Prediction expressed in Percentage of lifecycle

| Metric | Libgdx | Apache Struts | Hibernate | Spring |
|---|---|---|---|---|
| Complexity | 54.28% | 49.04% | 62.11% | 27.85% |
| Cohesion | 45.48% | 30.63% | 41.81% | 48.73% |
| Coupling | 59.69% | 32.46% | 67.01% | 46.94% |
| Inheritance | 49.30% | 47.41% | 53.17% | 61.45% |
| **Total** | **72.58%** | **55.87%** | **69.41%** | **68.32%** |

### 5.3. Comparative analysis against Maintainability Index (MI)

In an effort to further evaluate the effectiveness of our maintainability evaluation system in terms of providing accurate results, we compared it against the widely used Maintainability Index (MI). MI corresponds to a value between 0 and 100 that represents the relative ease of maintaining the source code. Higher values denote higher degree of maintainability. MI is computed using the following formula [32]:

$$\frac{171 - 5.2 \cdot HV - 0.23 \cdot CC - 16.2 \cdot \log(LOC)}{171} \cdot 100 \quad (5)$$

In the above equation, HV refers to Halstead Volume, CC to the McCabe Cyclomatic Complexity and LOC to the lines of code. According to [32], the MI values involve three different risk intervals. Values between 0 and 10 indicate high risk, while values from 10 to 20 indicate moderate risk. Values above 20 indicate that the source code is maintainable.

In order to perform our comparative analysis, we evaluated three different versions (releases) of the libgdx project in order to identify non-maintainable packages. The selected versions (`1.0.0`, `1.5.3`, and `1.9.9`) lie in different parts of the project lifecycle and thus we consider them suitable for evaluating the effectiveness of our methodology in a set of diverse assessment scenarios. Version `1.0.0` is the first major release of the project and lies in the beginning of its lifecycle, which indicates that the core functionality has been implemented. Version `1.5.3` is almost in the middle of the project lifecycle and refers to a typical timeframe that involves the development of new features, while maintaining already existing ones. Finally, version `1.9.9` is the most recent release (several releases ahead of version `1.5.3`), which indicates that the project has increased both in terms of size and complexity and thus is prone to facing maintainability issues.

Table 10: Non-Maintainable Packages Identification Strategies

| Strategy | Non-maintainability Criterion |
|---|---|
| **#1** | Average MI of classes included in the package is less than 10 |
| **#2** | Average MI of classes included in the package is less than 20 |
| **#3** | The package contains at least one class with MI value less than 10 |
| **#4** | The package contains at least one class with MI value less than 20 |

Given that maintainability index (MI) is computed at class level, we first compute the MI values of all classes included in the three aforementioned releases under evaluation. The computation was based on (5), but instead of using the values of Cyclomatic Complexity (CC) metric, we used the values of Weighted Methods per Class (WMC) metric (computed at class level), which corresponds to the sum of the CC values of the local methods and init blocks of the class. The next step involves using these values in order to identify non-maintainable packages and compare the results with the ones provided by our maintainability evaluation models. Based on the interperetation of MI values as given by [32], we employ two thresholds (10 and 20) in order to create two scenarios that differ in terms of strictness. In addition, given that MI is computed at class level (as opposed to our methodology, which is applied at package level) and in an effort to perform a comparison that does not favour our methodology, we generated two different strategies regarding each scenario. The first considers that a package is non-maintainable even if one of its classes is considered as non-maintainable (using the respective threshold), while the second uses the average MI value of its classes. As a result, we compare the results of our maintainability evaluation approach against MI following four different strategies. Table 10 presents these four strategies along with the respective criterion that leads to identify a package as non-maintainable.

Figure 9 illustrates the results obtained by applying the aforementioned strategies towards assessing the maintainability degree of the packages included in the three releases of the libgdx project along with the ones provided by our maintainability evaluation system. Each bar reflects the percentage of the correctly identified packages as non-maintainable. The blue bars refer to the strategies that employ MI, while the red ones refer to the results of our system. The results regarding all three releases indicate that our system is proven to be more efficient as compared to each strategy that employs MI values in order to identify the packages that are considered as non-maintainable. In addition and given the results among the different releases, it is obvious that our approach is getting more efficient as the analysis targets a later release (the percentage of correctly identified packages is constantly increasing), while all strategies based on MI appear to exhibit almost the same results regardless of the release under evaluation. This is expected as our modeling strategy is incremental, while MI uses a certain snapshot of the source code.

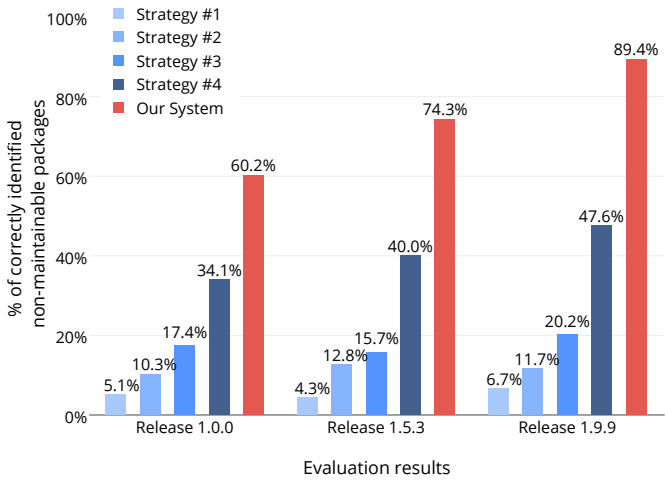



Figure 9: Comparative results regarding the identification of non-maintainable packages vs MI

The aforementioned conclusions can also be drawn by the inspection of the values of the MI metric for every class included in the non-maintainable packages. Figure 10 depicts the density of the MI values. Given that the MI values that denote the occurrence of non-maintainable software is less than 20, the expected outcome would be a distribution with the center closer to zero. Instead, the figure presents that the MI values are almost normally distributed within the range [10, 80]. As a result, MI does not appear to be very effective for identifying non-maintainable software.

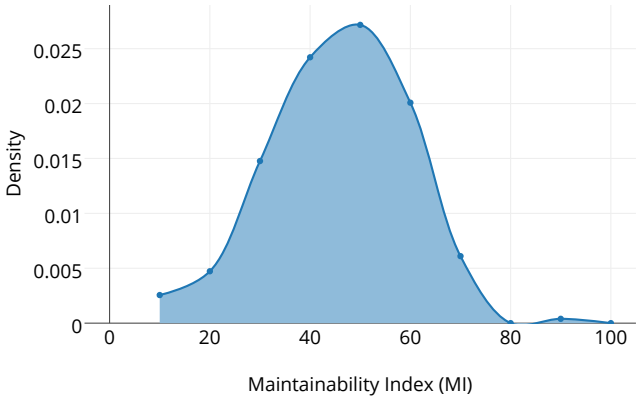



Figure 10: Maintainability Index density plot for methods included in non-maintainable packages

## 6. Threats to Validity

The limitations and threats to validity of our maintainability evaluation approach span along the following axes: a) limitations imposed by using the dropping of packages as a non-maintainability indicator, and b) choice of the aggregation strategy in order to compute the values of the static analysis metrics at package level.

Our approach involves using release information in order to identify the non-maintainable packages that will constitute the information basis upon which our maintainability evaluation models will be built. To that end, we assume that in cases a package is removed from a software system, then this is a non-maintainability indicator. Of course, removing a package from a software system may originate from other reasons that are not related to non-maintainability. For instance, a package may be dropped based on obsolete functionality or it can be simply renamed given a change in the naming conventions followed within the project. In an effort to minimize the impact of the aforementioned limitations, we applied a number of quality-related criteria. A dropped package is considered as non-maintainable only if at least one of its metrics has a trend that negatively affects its maintainability degree. In addition, towards capturing the renamed packages, we applied code clone detection techniques in order to identify the cases when two packages are identical and thus exclude them from our analysis. As a result, the introduced biases are limited to the ones that refer to package removals based on purely obsolete functionality.

Another threat to validity that occurs from our methodology is the fact that we aggregate the values of static analysis metrics, applicable at class level, in order to compute them at package level. During this process, we assume that each class contributes equally in the maintainability degree of its parent package and thus we selected simple average as our aggregation formula. As a result, in certain cases (application-specific), when there is available information that can be used to quantify the significance of each class in the maintainability degree of the parent package a different aggregation formula should be selected (e.g. weighted average using the significance metric as the weighting factor).

## 7. Conclusions and Future Work

In this work we propose a maintainability evaluation approach at package level depending on a releases analysis of the software project under evaluation. Our designed system is able to provide information regarding the extent to which each package of a given repository is maintainable by assessing the progressing behavior of four primary source code properties. The evaluation is built upon the trends of certain metrics each belonging to one of the four properties under evaluation: *complexity*, *cohesion*, *coupling*, and *inheritance*. As derived from the evaluation of the main features of our system, both qualitatively and quantitatively, we conclude that it cannot only accurately identify non-maintainable packages, but can also provide correct predictions at an earlier stage of the development process. This earlier stage is in many cases around 50% of the lifecycle which enables developers to fix the problematic behavior when less refactoring is needed and thus save both time and resources which results in reducing the software development cost. Another important feature of our system, unlike other maintainability evaluations systems, is its ability to provide interpretable results giving the insight to developers regarding the specific aspects of their code that are in need of improvement. Finally, we compared our assessment methodology with the widely-used maintainability index metric and the results indicate that our system was able to provide better results in terms of identifying non-maintainable source code. Considering all the above, we argue that our system can be a valuable tool for developers.

Future work relies in several directions. At first, we could further investigate the selection of metrics to be used for training each model. Additionally, we could also expand our trend analysis procedure by investigating more trend types especially non-linear in order to identify more progressing behaviors and thus model more complex maintainability evaluation scenarios. Furthermore, another interesting direction would be to expand the applicability of our methodology by introducing additional properties under evaluation that cover the needs and the special characteristics of additional programming languages (such as the non object-oriented). Finally, we could further explore the effectiveness of our methodology when applied upon ground

truth that employs different levels of granularity (other than releases i.e. commits) toward modeling the behavior of metrics.

## References


[1] ISO/IEC 25010:2011, https://www.iso.org/obp/ui/iso:std:iso-iec:25010:ed-1:v1:en, 2011. [Online; accessed July 2019].

[2] C. Faragó, P. Hegedus, G. Ladányi, R. Ferenc, Impact of version history metrics on maintainability, in: Proceedings of the IEEE 8th International Conference on Advanced Software Engineering & Its Applications (ASEA), 2015, pp. 30–35.

[3] G. A. Hall, J. C. Munson, Software evolution: code delta and code churn, Journal of Systems and Software 54 (2000) 111–118.

[4] M. Alshakhouri, J. Buchan, S. G. MacDonell, Synchronised visualisation of software process and product artefacts: Concept, design and prototype implementation, Information and Software Technology 98 (2018) 131 – 145. doi:https://doi.org/10.1016/j.infsof.2018.01.008.

[5] R. van Solingen, E. Berghout, R. Kusters, J. Trienekens, From process improvement to people improvement: enabling learning in software development, Information and Software Technology 42 (2000) 965 – 971. doi:https://doi.org/10.1016/S0950-5849(00)00148-8.

[6] J. C. Munson, S. G. Elbaum, Code churn: A measure for estimating the impact of code change, in: Proceedings of the IEEE International Conference on Software Maintenance, 1998, pp. 24–31.

[7] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change bursts as defect predictors, in: Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), 2010, pp. 309–318.

[8] S. Chidamber, C. Kemerer, A metrics suite for object oriented design., IEEE Transactions on Software Engineering 20 (1994) 476 – 493.

[9] G. M. Berns, Assessing software maintainability, Communications of the ACM 27 (1984) 14–23.

[10] P. Oman, J. Hagemeister, Construction and testing of polynomials predicting software maintainability, Journal of Systems and Software 24 (1994) 251–266.

[11] D. Coleman, D. Ash, B. Lowther, P. Oman, Using metrics to evaluate software system maintainability, IEEE Computing Practices 27 (1994) 44–49.

[12] K. D. Welker, The software maintainability index revisited, The Journal of Defense Software Engineering 14 (2001) 18–21.

[13] B. Seref, O. Tanriover, Software code maintainability: A literature review, International Journal of Software Engineering & Applications (IJSEA) 7 (2016) 69–87.

[14] R. Malhotra[1], A. Chug, Software maintainability prediction using machine learning algorithms, Software Engineering: An International Journal (SEIJ) 2 (2012) 19–36.

[15] T. G. Diamantopoulos, K. Thomopoulos, A. L. Symeonidis, Qualboa: Reusability-aware recommendations of source code components, 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR) (2016) 488–491.

[16] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, H. C. Almeida, Identifying thresholds for object-oriented software metrics, Journal of Systems and Software 85 (2012) 244–257.

[17] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–10.

[18] M. Foucault, M. Palyart, J.-R. Falleri, X. Blanc, Computing contextual metric thresholds, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 1120–1125.

[19] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using roc curves, Journal of Software: Evolution and Process 22 (2010) 1–16.

[20] V. Dimaridou, A.-C. Kyprianidis, M. Papamichail, T. Diamantopoulos, A. Symeonidis, Assessing the user-perceived quality of source code components using static analysis metrics, in: International Conference on Software Technologies, Springer, 2017, pp. 3–27.

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

[22] C. Jin, J.-A. Liu, Applications of support vector mathine and unsupervised learning for predicting maintainability using object-oriented metrics, in: 2nd International Conference on Multimedia and Information Technology (MMIT), volume 1, 2010, pp. 24–27.

[23] A. Kaur, K. Kaur, R. Malhotra, Soft computing approaches for prediction of software maintenance effort, International Journal of Computer Applications 1 (2010) 69–75.

[24] Y. Zhou, H. Leung, Predicting object-oriented software maintainability using multivariate adaptive regression splines, Journal of Systems and Software 80 (2007) 1349–1361.

[25] M. Papamichail, T. Diamantopoulos, A. Symeonidis, User-perceived source code quality estimation based on static analysis metrics, in: IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 100–107.

[26] I. Samoladas, I. Stamelos, L. Angelis, A. Oikonomou, Open source software development should strive for even greater code maintainability, Communications of the ACM 47 (2004) 83–87.

[27] F. Fioravanti, P. Nesi, Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems, IEEE Transactions on software engineering 27 (2001) 1062–1084.

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

[29] Y. Kanellopoulos, C. Tjortjis, I. Heitlager, J. Visser, Interpretation of source code clusters in terms of the iso/iec-9126 maintainability characteristics, in: 12th European Conference on Software Maintenance and Reengineering (CSMR), 2008, pp. 63–72.

[30] R. Harrison, S. J. Counsell, R. V. Nithi, An evaluation of the mood set of object-oriented software metrics, IEEE Transactions on Software Engineering 24 (1998) 491–496.

[31] Maintainability Realated Metrics, https://maintainability-metrics.netlify.com/, 2011. [Online; accessed July 2019].

[32] Maintainability Index, https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/, 2007. [Online; accessed July 2019].