# Assessing Change Proneness at the Architecture Level: An Empirical Validation

Elvira-Maria Arvanitou[1], Apostolos Ampatzoglou[1], Konstantinos Tzouvalidis[2], Alexander Chatzigeorgiou[3], Paris Avgeriou[1], Ignatios Deligiannis[2]

[1] Department of Computer Science, University of Groningen, Netherlands
[2] Department of Computer Science, Technological Education Institute of Thessaloniki, Greece
[3] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

e.m.arvanitou@rug.nl, a.ampatzoglou@rug.nl, kostastzouvalidis@gmail.com, achat@uom.gr, paris@cs.rug.nl, ignatios@it.teithe.gr

*Abstract*—**Change proneness is a characteristic of software artifacts that represents their probability to change in future. Change proneness can be assessed at different levels of granularity, ranging from classes to modules. Although change proneness can be successfully assessed at the source code level (i.e., methods and classes), it remains rather unexplored for architectures. Additionally, the methods that have been introduced at the source code level are not directly transferrable to the architecture level. In this paper, we propose and empirically validate a method for assessing the change proneness of architectural modules. Assessing change proneness at the level of architectural modules requires information from two sources: (a) the history of changes in the module, as a proxy of how frequently the module itself undergoes changes; and (b) the dependencies with other modules that affect the probability of a change being propagated from one module to the other. To validate the proposed approach, we performed a case study on five open-source projects. Specifically, we compared the accuracy of the proposed approach to the use of software package metrics as assessors of modules change proneness, based on the 1061-1998 IEEE Standard. The results suggest that compared to examined metrics, the proposed method is a better assessor of change proneness. Therefore, we believe that the method and accompanying tool can effectively aid architects during software maintenance and evolution.**

*Keywords*—*Architectural metrics; Change proneness; Empirical*

## I. INTRODUCTION

Change proneness is defined as the susceptibility of an artifact to change in an upcoming versions of a system [14], and is a cornerstone of change impact analysis [10]. Change proneness can be defined, quantified, and assessed on artifacts from different development phases, e.g., at the implementation level for assessing the urgency to eliminate the existence of a code smell [7], or at the architecture level for exploring the ripple effects along maintenance [2]. An application of change proneness at architecture level[1], is its use as a proxy of interest probability for architectural technical debt [19]. Specifically, it is claimed that the repayment of technical debt for architectural modules should be prioritized considering their susceptibility to change. In other words, inefficiencies identified in mod-ules, do not pose a serious risk regarding projects' sustainability, when these modules are not frequently maintained / modified [19]. Moreover, identifying modules that are change prone can steer test planning, by focusing on parts of the architecture that are more likely to undergo changes due to maintenance.

In the literature, one can identify several approaches for assessing class change proneness (see Section II), but no approach at the architecture level and specifically on the level of architectural modules. According to a recent mapping study on design-time quality attributes, change proneness (and its related quality attribute, namely instability) has been assessed by eight studies at the implementation level (e.g., [6]), six at the detailed-design level (e.g. [18]), but none at the architecture level [4]. *Despite the existence of many methods on assessing the change proneness of artifacts at the implementation and design level, these methods are not directly transferable to the architecture level.* Such a transfer would require, either: (a) the aggregation of the class level measurements to module level, using some known function (e.g., average, maximum, etc.), or (b) the re-introduction of the method's constructs to the architecture level. The option of using aggregation functions is not considered optimal, because it could potentially lead to inaccurate results. For example (see Fig. 1), consider the assessment of a two-package relationship (A→B) where each one contains five classes (A1-A5 and B1-B5, respectively) and the packages communicate only through one interface (e.g., assume that A1 calls methods from B1).
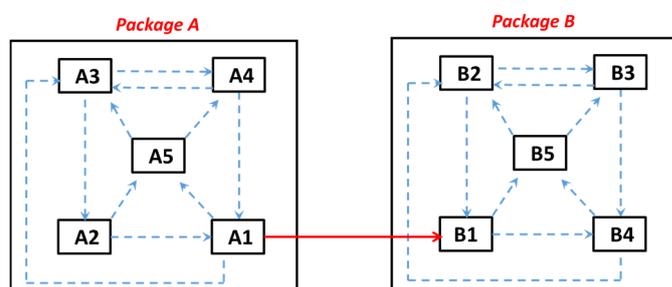


Fig. 1. *Aggregation of metrics to the architecture level*

As expected, classes belonging to the same package collaborate to serve their common purpose (high intra-module cohesion), therefore they are coupled to each other (e.g. each class

---

[1] In the rest of the paper, the term 'architecture level' refers to the level of architectural modules. To scope our study further, as architectural modules, we consider source code packages, which are collections of classes.

communicates with two others). The use of average would lead to an aggregated efferent coupling (Ce) [13] at the package level of 2.2 (A1: 3, A2-A5: 2), and a sum coupling that equals 11. However, this metric is inaccurate at the module level since the only inter-module dependency that exists is between A1 and B1. Thus, the option *to reshape a method to fit the architecture level* is expected to yield more accurate results. However, this approach essentially leads to a new method that needs to be evaluated from scratch, so as to validate its fitness in the context of architecture.

In this paper, with the goal to provide a change proneness assessment method for architectural modules, we proceed with the option to tailor the constructs of a method assessing change proneness at the design level (i.e., [3]) at the level of architecture. Based on the original method, to calculate the change proneness of an artifact, two parameters need to be quantified [17]: (a) *the history of changes of the artifact*, which can be captured e.g., through the frequency of changes along evolution; and (b) *the structural characteristics of the software*, such as coupling [3],[5]. To this end, we propose how these two parameters can be quantified (or at least assessed) by considering architectural modules (i.e., packages[1]—a collection of classes).As an outcome, the updated method calculates a metric, namely Module Change Proneness Measure (MCPM). Additionally, for the reasons explained before, we empirically validate the accuracy of the derived model, by comparing its validity with existing architectural coupling metrics. The evaluation is performed on five large-scale Open Source Software (OSS) projects that provide us with 160 modules as units of analysis. The rationale and the study setup for the proposed method is a replication of the evaluation method proposed in the original study [3]. The evaluation is performed empirically, based on the guidelines of the 1069-1998: IEEE Standard on Software Measurement.

The rest of the paper is organized as follows: In Section II we discuss related work and background information on metric validation guidelines, whereas in Section III we present the proposed method for quantifying module change proneness. Section IV presents the design of the case study, whereas its results are presented in Section V. In Section VI we discuss the main findings of validation. Finally, Sections VII and VIII present threats to validity and conclude the paper.

## II. BACKGROUND INFORMATION

In this section we discuss research efforts related to change proneness assessment at minimum on design level (see Section II.A) and metric validation criteria as defined by the 1069-1998: IEEE Standard on Software Measurement (see Section II.B)

### A. Related Work

In the early '80s Yau and Collofello suggested the first measures for design instability (i.e., a term that is conceptually relevant to change proneness). Both measures were considering the probability of an actual change to occur, the complexity of the changed module, the scope of the used variables, and the relationships between modules [18]. However, the specific studies (they are among the first ones that discuss instability as a quality attribute) are kept at a rather abstract level, without

proposing specific metrics or tools for quantifying them. In a more recent study, Black proposed an approach for instantiating the theoretical approach of Yau and Collofello, by calculating a model for assessing module change proneness. The approach calculates complexity, coupling, and control flow metrics, and their combination provides an estimate of change proneness [6]. The difference of the work of Black compared to our study is that Black considers single file as modules, which is at a lower level of granularity than the package level.

Additionally, many researchers have assessed change propagation at the class level. For instance, Han et al. proposed a metric that can be used for assessing change proneness of classes, based on studying the behavioral dependencies of classes [9]. Similarly, Lu et al. conducted a meta-analysis to investigate the ability of object-oriented metrics to evaluate change proneness [11]. The results suggested that size metrics are the optimum assessors of change proneness, followed by cohesion and coupling metrics [11].Finally, Schwanke et al. dealt only on bug-related change frequency (i.e., fault proneness), and tried to identify assessors for it [16]. The results of this study proposed that fan-out (i.e., the number of other artifacts which a module depends on) is a good assessor of change proneness [16], further highlighting the appropriateness of coupling metrics as assessors of modules' change proneness.

### B. Metric Validation Criteria

For comparing the validity of MCPM to existing coupling metrics, we will use the criteria described in the 1061 IEEE Standard for Software Quality Metrics [1]. In this standard, six metric validation criteria are suggested, accompanied by the statistical test that shall be used for evaluating every criterion:

- *Predictability* assesses the accuracy with which the metric under study applied at a time point t1 is able to predict the levels of the quality characteristic at a time point t2. The criterion is quantified through the standard estimation error for a regression model [1].

- *Discriminative Power* assesses if the metric under study is capable of discriminating between high-quality and low-quality components. Discriminative power can be quantified through a contingency table [1]. However, this type of quantification was not applicable for coupling metrics, because they cannot be recoded as categorical variables, without setting arbitrary thresholds. Therefore, we use the Kruskall-Wallis test [8].

- *Correlation* assesses the relationship between a quality characteristic and the metric under study to confirm that the use of the metric can substitute the characteristic. The criterion is quantified by using a correlation coefficient [1].

- *Consistency* assesses whether there is consistency between the ranks of the quality characteristic and the ranks of the metric under study. Consistency determines if a metric can accurately rank artifacts in terms of quality. The criterion is quantified by the coefficient of rank correlation [1].

- *Tracking* assesses if values of the metric under study can follow changes in the levels of the quality characteristic. Tracking is quantified by using the coefficient of rank correlation for a set of project versions [1].

- **Reliability** assesses if the studied metric can fulfill all the aforementioned criteria, in a sufficient number of projects. Reliability offers evidence that a metric can perform its intended function consistently. Reliability can be assessed by replicating the previously discussed tests (for each of the aforementioned criteria) to various software systems [1].

## III. Proposed Method

The probability that a module (in our case a package, i.e. a set of classes) will change in the future is affected not only by the likelihood of modifying the module itself, but also by possible changes in other modules that might propagate to it. Thus, the calculation of change proneness is based on two main factors: **the internal probability to change** (i.e., the probability of a module to change due to changes in requirements, bug fixing, etc.) and **the external probability to change**, which corresponds to the probability of a module to change due to ripple effects (i.e., changes propagating from other modules). To calculate the external probability to change, the various dependencies between modules need to be considered: if module A has a dependency to module B, the external probability of A to change due to B is obtained as:

$$P(A:external_B) = P(A|B) \cdot P(B)$$

P(A|B) is the **propagation factor** between module B and A (i.e., the probability that a change made in B is emitted to A). P(B) refers to the **internal probability** of changing module B.

To illustrate our method, let's consider the example of Fig. 2, depicting four packages and some of the contained classes as well as the dependencies among packages, which in turn are due to dependencies between the contained classes.
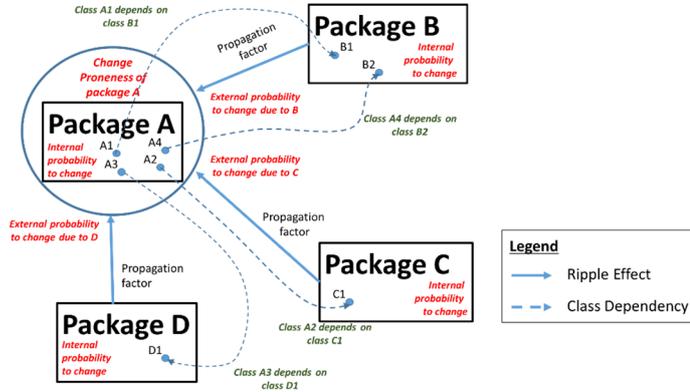


Fig. 2. *Example System for MCPM Demonstration*

The calculation of change proneness for module A (see Fig. 2) should take into account the:

- *Internal probability* to change of A—P(A). This probability refers to the likelihood of changing any of the classes contained in package A for the resolution of bugs or the introduction of a novel feature.
- *External probability* to change due to ripple effects from package B—P(A:external_B).The value corresponds to the probability of A to change because of its dependency to

B. It depends on the internal probability of B to change (as a trigger to the ripple effect) and the possibility of changes to propagate through the B→A dependency (as a proxy of the probability that the change be emitted).

- *External probability* to change due to ripple effects from package C—P(A:external_C).
- *External probability* to change due to ripple effects from package D—P(A:external_D).

Since a module might be involved in several dependencies and because even one change in the dependent modules will be a reason for changing that module, the module change proneness measure (MCPM) is calculated as the joint probability of all events that can cause a change to a module. In this example, module A might change due to the following events: (a) change in A itself, (b) a ripple effect from B, (c) a ripple effect from C, or (d) a ripple effect from D, as follows:

$$\textbf{MCPM(A)} = \text{Joint Probability}\{P(A), P(A:external_B), P(A:external_C), P(A:external_D)\}$$

The accuracy in assessing MCPM depends on the precision of the estimates of the internal probability of change for each module and the propagation factor for each dependency. Regarding **the internal probability of change** we use the percentage of commits in which a module has changed [20]. We study all commits between two successive versions of a system and count in how many of those, at least one class of the module has changed. This percentage is calculated for all past pairs of versions, and the obtained average is used as the internal probability of change. Concerning **the propagation factor of changes** among dependent modules we tailored the Ripple Effect Measure (REM) [3], which quantifies the probability of a change occurring in class B to be propagated to a dependent class A. REM essentially quantifies the percentage of the public interface of a class that is being accessed by a dependent class. The calculation of REM is based on dependency analysis. Such change propagations [3], are the result of certain types of changes in one class (e.g., a change in the method signature—i.e., method name, types of parameters and return type—that is invoked inside another method) that potentially emit changes to other classes. To fit the architecture level, REM has been changed to deal with modules instead of classes. At the module level we consider all class dependencies that reach across modules. For example, to calculate the REM from package B to package A in Fig. 2, we consider two dependencies, namely A1 to B1 and A4 to B2. The aggregation of class to module dependencies yielding the Ripple Effect Measure between packages B and A, REM(B→A), is performed as follows, by tailoring the original definition of REM:

$$REM(B \rightarrow A) = \sum_{i=0}^{i < dependencies(A \rightarrow B)} \frac{NPrA(Bi) + NOP(Bi) + NDMC(Ai \rightarrow Bi)}{NOM(Bi) + NOA(Bi)}$$

NDMC: **number of direct method calls**
NOM: **number of methods**
NOA: **number of attributes**
NPrA: **number of protected attributes** (only for inheritance)
NOP: **number of polymorphic methods** (only for inheritance)

Details of REM calculation are provided in the paper, in which we defined and validated it [3].The rationale for building the REM calculation formula can be summarized as follows: The ratio of the two aforementioned counts is an estimate of the probability that a random change in the public interface of source class will occur in a member that will emit this change to the dependent class. In other words, as the number of the members of the source class that emit changes to another dependent class, approaches the total number of members that can change in the source class, it becomes more probable for changes to propagate from the source class to the dependent class. Based on REM definition, the formula for calculating external probability, presented before, is updated as follows:

$$P(A:external_B) = REM(B{\to}A){\bullet}P(B)$$

As any other probability the range of MCPM is [0, 1] (i.e., 0% to 100%). Although, we are not able to provide a threshold that discriminates highly from low change prone modules, the metric can prove useful for comparison purposes.

## IV. CASE STUDY DESIGN

To investigate the validity of MCPM as an assessor of change proneness, we performed a case study on five OSS projects, and compare MCPM to three package-level coupling metrics. Coupling metrics have been considered in this study for two reasons: (a) they represent the existence / strength of dependencies among modules, and are thus structural metrics that can be considered as a proxy of external probability of change; and (b) they are reported in related studies (e.g. [16] and [18]) as fair assessors of change proneness. By considering that this study focuses on the architecture level, we needed to identify metrics calculated at module or package level [13]:

- *Afferent Coupling (Ca)*—the number of classes in other packages that depend upon classes within the package. An indicator of packages responsibility. Afferent couplings signal inward dependencies [13];

- *Efferent Coupling (Ce)* —the number of classes in other packages that at least one class in a package depends upon. An indicator of packages dependence on external modules. Efferent couplings signal outward dependencies that consist reasons for change [13]; and

- *Instability (I)* —the ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of packages' resilience to change, and its range is [0, 1]: I=0 indicating a completely stable package and I=1 indicating an unstable package [13].

The study has been designed and reported according to the guidelines of Runeson et al. [15]. In this section, we present: (a) the goal of the case study and the derived research questions, (b) the description of cases and units of analysis, (c) the data collection, and (d) the process for data analysis.

### A. Objectives and Research Questions.

This study aims *to **analyze** MCPM and package metrics **for the purpose of** evaluation **with respect to** their validity to assess module change proneness, **from the point of view of** architects **in the context of** software maintenance and evolution.* Based this goal, we have set two research questions:

**RQ₁**: How does MCPM compare to package metrics with respect to their validity as assessors of change proneness based on the IEEE Standard on Software Measurement (i.e., predictability, discriminative power, correlation, consistency, and tracking)?

**RQ₂**: How does MCPM compare to package metrics with respect to reliability?

RQ₁ aims to investigate the validity of the proposed measure, in comparison to three existing metrics, with respect to the first five validity criteria (i.e. correlation, consistency, tracking, predictability and discriminative power). For this research question we employ a single dataset comprising all examined projects. RQ₂ aims to investigate the validity in terms of reliability. Reliability is examined separately since, according to its definition, each of the other five criteria should be tested on different projects. For RQ₂we consider each project as a different dataset and then results are cross-checked to assess metrics' reliability.

### B. Case Selection Units of Analysis and Selection

This study is an embedded multiple-case study, i.e., it studies multiple cases and each case is comprised of many units of analysis. Specifically, the cases are open source projects, whereas the units of analysis are their packages (i.e., the reporting is performed at the project level). The results are aggregated to the complete dataset by using the percentage of projects in which the results are statistically significant. As subjects we selected to use the last five versions of five open source software (OSS) projects. A short description of the goals of these projects is provided in Table I, along with some demographics.

TABLE I. PROJECT DEMOGRAPHICS

| Project | Training Transitions | Assessment Transitions | Description |
|---|---|---|---|
| wro4j (32 packages) | 1.7.0 → 1.7.8 | 1.7.8 → 1.8.0 | a tool for optimization of web resources |
| Guava (17 packages) | 11.0 → 19.0 | 19.0 → 21.0 | a set of libraries for new collection types |
| commons-lang (12 packages) | 3.0.1 → 3.3.2 | 3.3.2 → 3.5 | a host of helper utilities for the java language API |
| joda-time (7 packages) | 2.8.2 → 2.9.7 | 2.9.7 → 2.9.9 | a replacement for the Java date and time classes |
| Wicket (72 packages) | 7.0.0 → 8.0.0.2 | 8.0.0.2 → 8.0.0.4 | a web application framework |

The projects have been selected based on: (a) *their popularity*—i.e., highly reused libraries and frameworks (according to Maven Repository), (b) *their programming language*—the used tools can only parse Java code, (c) *their non-trivial size*—i.e., more than 500 classes (although their number of packages differs), and (d) *their consistency in releasing new versions in rather stable timeframes*—this is important since the training versions should have a similar number of commits as the assessment versions. Thus, our study was performed on about 160 Java packages (on average: ~30 per project).

## C. Data Collection& Analysis

For each unit of analysis (i.e., package), we recorded eight variables: (a) *Demographics*—project, version, package name; (b) *Assessors (MCPM, Ca, Ce, and I)*—these variables are going to be used as the independent variables, and are calculated in the last training version; and (c) *Actual changes*—we calculate the percentage of commits in which the corresponding package has changed (PCPC) in the transition between the last two versions of a system (i.e., those that we want to assess—see last column of Table I), as the variable that captures the actual changes. PCPC is going to be used as the dependent variable in all tests, representing the actual change proneness. The aforementioned metrics have been calculated with two tools. PCPC is calculated by a tool that uses the GitHub API to count in how many commits each package has been modified[2].All assessors have been calculated by modifying the tool of Tsantalis et al. [17].

The variables are analyzed against the criteria of 1061 IEEE Standard, as imposed by the standard per se. More details on the assessment of each criterion are provide in Section II, whereas an overview is presented in Table II.

TABLE II.   MEASURE VALIDATION ANALYSIS

| Criterion | Test | Variables | Target Version |
|---|---|---|---|
| Predictability | Linear Regression | **Independent**: Assessors **Dependent**: Actual Changes | *Last* |
| Discriminative Power | Kruskal-Wallis Test | **Testing**: Assessors, **Grouping**: Actual Changes | *Last* |
| Correlation | Pearson Correl. | **Independent**: Assessors **Dependent**: Actual Changes | *Last* |
| Consistency | Spearman Correl. | | *Last* |
| Tracking | Spearman Correl. | | *All* |
| Reliability | All the aforementioned tests | | *All* |

## V.   RESULTS

In this section, we present the results of the case study. Section V.A presents the results on comparing MCPM to other candidate change proneness assessors, with respect to five criteria (correlation, tracking, consistency, predictability and discriminative power), and Section V.B concerns reliability.

## A.  Correlation, Consistency, Tracking, Predictability and Discriminative Power (RQ1)

In Table III we present the results of the univariate Linear Regressions that have been performed to validate the predictive power of each assessor; and in Table IV the results on the Discriminative Power of the assessors. The cells of Table III represent the standard error of the regression model and the cells of Table IV represent the level of significance in the differences of metric scores. The rest of the notations remain unchanged. Table III suggests that MCPM and Ce are the best predictors of package change proneness. In addition, the results of Table IV suggest that MCPM and Ce are the optimal assessors for discriminating groups of packages, based on their change proneness, i.e., classify them into groups with similar values of change proneness.

TABLE III.   PREDICTIVE POWER

| Project | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| wro4j | **.030** | .031 | **.030** | .032 |
| Guava | *.104* | *.110* | .115 | .119 |
| commons-lang | *.067* | .112 | *.075* | .125 |
| joda-time | **.319** | .324 | .320 | .320 |
| Wicket | *.012* | *.012* | **.008** | .013 |
| % sig. | **60%** | 40% | 40% | 0% |

TABLE IV.   DISCRIMINATIVE POWER

| Project | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| wro4j | *.008* | .587 | *.049* | .613 |
| Guava | **.059** | .277 | .139 | .835 |
| commons-lang | .999 | **.727** | .999 | .889 |
| joda-time | .381 | **.190** | .571 | .381 |
| wicket | *.000* | .734 | *.000* | .862 |
| % sig. | **40%** | 0% | **40%** | 0% |

In Tables V - VII, we present the results of the first three criteria: (a) correlation, (b) consistency, and (c) tracking. Each row of the tables represents one project, whereas each column denotes the correlation coefficient for each metric (i.e., Pearson for correlation and Spearman for consistency). Regarding tracking (see Table VII) the cells of each column present the mean Spearman correlation coefficient obtained by assessing the change proneness for all versions. The italic fonts denote statistically significant correlations, whereas bold fonts the assessor that is the most highly correlated with actual change proneness. Finally, the last row of each table corresponds to the percentage of projects, in which the specific assessor is significantly correlated to the actual change proneness.

TABLE V.   CORRELATION ANALYSIS

| Project | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| wro4j | *.348* | .288 | .346 | .102 |
| Guava | **.487** | .407 | .272 | .109 |
| commons-lang | **.805** | .156 | *.754* | -.166 |
| joda-time | **.205** | .090 | -.187 | -.409 |
| Wicket | *.476* | *.412* | **.791** | -.016 |
| % sig. | **80%** | 40% | 40% | 0% |

Table V suggests that MCPM is in 60% of the cases strongly correlated (see interpretation of correlation coefficients in [12]—corr. coefficient > 0.4) to actual package change proneness. At the individual project level, MCPM is very strongly correlated to change proneness for 20% of the projects, strongly correlated for 40%, and moderately correlated for 40%; whereas it is the most valid assessor in terms of correlation for 80% of the projects. We note that MCPM is significantly correlated with change proneness in all OSS projects that we have examined, whereas Ca and Ce only in 40%. Additionally, MCPM is the best change proneness assessor in

all three criteria—see Tables V to VII. However, we observe that the results on tracking (Table VII) have lower values, denoting decreased validity when considering the complete project lifetime.
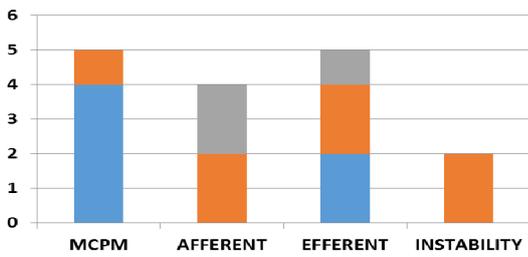
TABLE VI. CONSISTENCY ANALYSIS

| Project | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| wro4j | *.398* | .052 | *.379* | .110 |
| Guava | .437 | *.484* | .409 | .197 |
| commons-lang | **.306** | .110 | .242 | .013 |
| joda-time | *.378* | .321 | -.161 | -.400 |
| Wicket | *.419* | -.069 | *.623* | .059 |
| % sig. | **60%** | 20% | 40% | 0% |

TABLE VII. TRACKING ANALYSIS

| Project | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| wro4j | *.390* | .050 | *.375* | .105 |
| Guava | .400 | *.450* | .301 | .150 |
| commons-lang | **.301** | .106 | .240 | .010 |
| joda-time | **.370** | .317 | -.155 | -.395 |
| Wicket | *.410* | -.064 | *.618* | .052 |
| % sig. | **40%** | 20% | **40%** | 0% |

## B. Reliability (RQ$_2$)

Regarding RQ$_2$, we executed all the aforementioned tests separately for each project. For a metric to be considered a reliable assessor of change proneness, it should be consistently ranked among the top assessors for each criterion. To visualize this information, in Figures 3a - 3e we present a stacked bar chart for each validity criterion. In each chart, every bar corresponds to one change proneness assessor, whereas each stack represents the ranking of the assessor among the evaluated ones for each project.
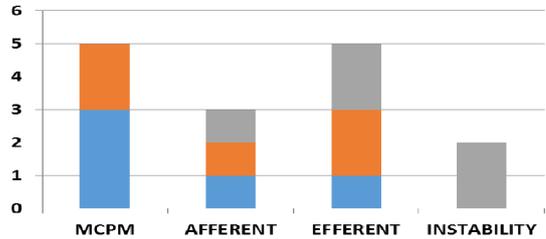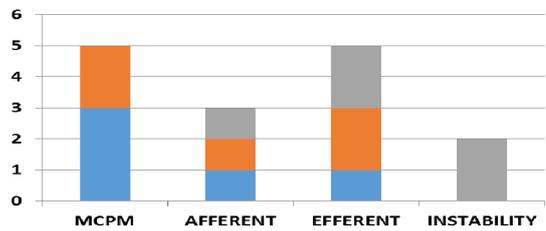

(a) Predictability Analysis
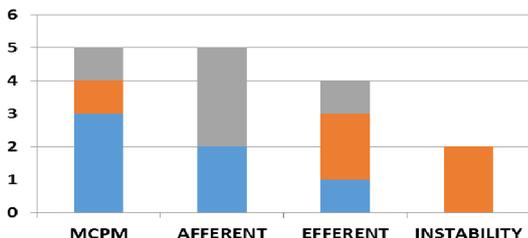

(b) Discriminative Power Analysis
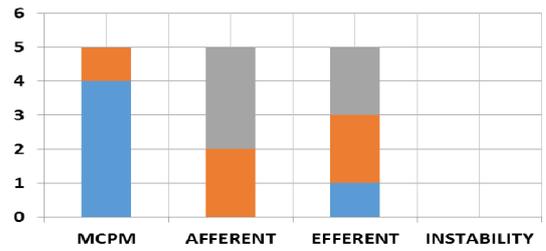

(c) Correlation Analysis


(d) Consistency Analysis


(e) Tracking Analysis

*Fig.3.Reliability Assessment*

From Figure 3c, we can observe that MCPM is the top-1 assessor (blue) of change proneness, with respect to correlation in three projects and the top-2 assessor (orange), for one other project. For some charts the count of 1st (blue), 2nd (orange) and 3rd (grey) positions does not sum up to five, since in case of equal scores, metrics are assigned the highest rank.

TABLE VIII. RELIABILITY ANALYSIS

| Criterion | MCPM | Ca | Ce | I |
|---|---|---|---|---|
| Corelation | **14** | 7 | 9 | 0 |
| Consistency | **13** | 6 | 9 | 2 |
| Tracking | **13** | 6 | 9 | 2 |
| Predictive Power | **14** | 6 | 11 | 4 |
| Discriminative Power | **12** | 9 | 8 | 4 |
| Total | **66** | 34 | 46 | 12 |

In Table VIII we present a synthesized view of the aforementioned results. Specifically, we use a point system to evaluate the consistency with which each assessor is highly ranked among others in all criteria. In particular, for every first position we reward the assessor with three points, for every second position with two points, and for every third position with one point. In Table VIII each row represents a criterion, whereas each column an assessor of change proneness. The cells represent the points that each assessor scored for each criterion. The

last row, presents a sum of all criteria. The results presented in both Table VIII and Figure 3, suggest that MCPM is the most reliable assessor of package change proneness, followed by Ce.

## VI. DISCUSSION

### A. Interpretation of the Results

The results of this study suggest that at the architecture level, MCPM is a better assessor of change proneness, compared to all other explored metrics, followed by Ce. It is expected that MCPM outperforms other metrics, mostly because it combines the two aspects of change proneness (i.e., probability of the package itself to change due to changes in requirements, bug fixes, etc. and the probability of a package to change due to the ripple effects), whereas all other coupling metrics consider only the second aspect (i.e., structural dependencies). In addition, all other package metrics are just counting dependencies among packages and do not quantify the strength of the relationship. The proposed measure considers the percentage of the public interface of a module that is being accessed by another module and therefore accurately captures the probability of change propagation between them.

Another interesting observation is that the validity of all metrics in terms of tracking is lower compared to consistency. This outcome is expected since the training set for assigning the value of the internal package probability is getting smaller, while we explore earlier project versions, and therefore the tracking ability becomes less accurate. This outcome implies that using a project history longer than five versions, might increase even more the validity of MCPM. However, this statement needs to be empirically evaluated by a follow-up study. Furthermore, by comparing the package metrics of this study, we can observe that efferent coupling is a better assessor of change proneness than afferent coupling and instability. This finding is reasonable since outward dependencies (i.e., packages in which a package relies upon) are more important than inward ones when assessing the susceptibility of modules to change. Additionally, this result is in accordance to related work, at the class level, which suggests that the fan-out metric is a more important parameter than fan-in regarding change proneness [16].

### B. Implications to Researchers and Practitioners

Based on the aforementioned results and discussions, we can provide implications for researchers and practitioners. On the one hand, we encourage practitioners, and especially architects, to use MCPM in their quality monitoring processes, in the sense that MCPM is a better assessor of the probability of a module to change, compared to other metrics (although a more thorough validation with practitioners is still required). We expect that tool support automating the calculation process will ease its adoption. Based on the expected relations of change proneness to more high-level quality characteristics (e.g., increased defect-proneness, more technical debt interest, etc.), it can be used as an assessor of future quality indicators. In particular, test case prioritization can highly benefit from observing the value of MCPM for system modules that are changing. For example, additional modules that need to be tested can be identified through the dependency analysis pro-

vided by the tool. The tool aids in test prioritization in the sense that it designates the probability of the dependent module to change due to ripple effects. We believe that the tailoring of the method to the architecture level consist it even more beneficial (compared to the class level), since it increases the scalability of change impact analysis to larger systems.

On the other hand, we suggest that researchers should tailor the MCPM to the level of requirements, i.e., assess the probability of a requirement to change in the future and compile a list of other requirements that might be affected. We believe that such a transformation would be of great interest for the software engineering community, in the sense that it could be used for test case prioritizing, adaptive maintenance activities, etc. Finally, we note that other claims that have already been stated in the manuscript that require further validation constitute interesting future work, i.e.: (a) the increase in the assessing power of MCPM when a larger portion of software history is considered as a training set for the method; (b) the usefulness of the proposed metric in practice and its adoption by practitioners; and (c) the validity of the MCPM metric in other levels of granularity.

## VII. THREATS TO VALIDITY

In this section we present the threats to the validity of our case study. Threats to **construct validity** [15] concern how metrics and change proneness are quantified, including both the rationale of the calculation and tool support. Concerning the rationale, we note that their definition is clear and well-documented (see Section III), whereas the used tools have been thoroughly tested, before deployment, in a large number of open source projects. Nevertheless, assessing the internal probability of a module to undergo changes on the basis of past changes has limitations as it cannot capture all potential reasons for future changes. Moreover, the probability of change propagation through static dependencies does not capture other, conceptually related, dependencies between modules. Finally, two additional threats to construct validity stem from the calculation of PCPC. In particular: (a) by calculating PCPC from all commits, without discriminating those occurring due to ripple effects, raises an issue, since the external probabilities to change are double counted in the model. However, since the current version of MCPM is validated as accurate enough, we preferred not to make its calculation even more complex, in the sense that such a discrimination would require manual inspection of all commits; (b) the evolution of a project might not be stable across all releases. For example, it is expected that in early stages of development, the changes are more massive, and become more focused as the project matures. Therefore, the PCPC changes significantly into these two stages.

The **reliability** of the present study concerns the replicability of the collected data and the performed analysis. To ensure the reliability [15] of this study, we: (a) thoroughly documented the study design in this work (see Section 4), to make the study replicable, and (b) all steps of data collection and data analysis have been performed by two researchers in order to prevent the introduction of bias. Additionally, the data analysis part is based solely upon statistical analysis (quantitative study), a fact that guarantees the elimination of any researcher

bias in terms of results interpretation. The low number of subjects (five OSS projects) is a threat to **external validity** [15], in the sense that results on these projects cannot be generalized to the complete OSS population. However, since the units of analysis for this study are packages and not projects, we believe that this threat is partially mitigated. Second, we investigated projects only written in Java due to the corresponding tool limitations. Therefore, the results cannot be generalized to other languages, e.g., C++. Moreover, we note that our results are not applicable for modules of non-object-oriented systems, since our definition of module applies only in this programming paradigm. Finally, our metric is not applicable for projects that are not hosted in version control management systems, since the calculation of PCPC requires access to the complete development history of the project.

## VIII. CONCLUSIONS

In this study, we presented and validated a new method that calculates the Module Change Proneness Metric (MCPM), which can be used for assessing the change proneness of software modules. The method takes inputs from two sources: (a) module dependencies, which are used to calculate the portion of the accessible interface of a module that is used by other modules, and (b) module change history, which is used as a proxy of how frequently maintenance actions are performed (e.g., modify requirements, fix bugs, etc.). After quantifying these two parameters (for all modules and for all their dependencies), MCPM can be calculated at the architecture level, by employing simple probability theory. In this work MCPM has been empirically validated against three other change proneness assessors, based on the criteria defined in the 1061-1998 IEEE Standard for a Software Quality Metrics [1]. The conducted case study was embedded, and was executed on five open source software projects, which in total offered us more than 160 units of analysis (i.e., software packages).

The results of the validation suggested that MCPM excels as an assessor of module change proneness compared to other coupling package metrics. In particular, the results implied that both the historical and the structural information are needed for an accurate assessment in the sense that the combined perspective provided by MCPM has been evaluated as the optimal assessor of change proneness, with respect to all validation criteria. Based on these results, implications for researchers and practitioners have been provided. More specifically, researchers are encouraged to tailor the proposed metric to fit the requirements level, whereas practitioners are encouraged to introduce the proposed metric in the quality dashboards or quality gates, in order to improve the maintainability of their source code and accurately perform test case prioritization.

## REFERENCES

[1] 1061-1998: IEEE Standard for a Software Quality Metrics Methodology, IEEE Standards, IEEE Computer Society, re-affirmed Dec. 2009).

[2] S. Anwar, F. Idris, M. Ramzan, A. A. Shahid and A. Rauf, "Architecture Based Ripple Effect Analysis: a Software Quality Maintenance Perspective", *10th International Conference on Information Science and Applications (ICISA' 10)*, IEEE Computer Society, pp. 1-8, Seoul, 2010.

[3] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Introducing a ripple effect measure: a theoretical and empirical validation", 9th International Symposium on Empirical Software Engineering and Measurement (ESEM '15), IEEE Computer Society, 22–23 October 2015, China.

[4] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou, "A Mapping Study on Design-Time Quality Attributes and Metrics", Journal of Systems and Software, Elsevier, 127 (5), pp. 52-77, May 2017.

[5] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "A Method for Assessing Class Change Proneness", 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, 15-16 June 2017, Sweden.

[6] S. Black, "Deriving an approximation algorithm for automatic computation of ripple effect measures", Information and Software Technology, Elsevier, 50, pp. 723-736, 2008.

[7] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Assessing Code Smell Interest Probability: A Case Study", 9th International Workshop on Managing Technical Debt (MTD' 17), ACM, 22 May 2017, Germany.

[8] A. Field, "Discovering Statistics using IBM SPSS Statistics", SAGE Ltd, 2013.

[9] A. R. Han, S. Jeon, D. Bae, and J. Hong, "Measuring behavioral dependency for improving change-proneness prediction in UML-based design models", Journal of Systems and Software, Elsevier, 83 (2), pp. 222-234, February 2010.

[10] F. M. Haney, "Module connection analysis: A tool for scheduling of software debugging activities", AFIPS Fall Joint Computer Conference, pp. 173-179, 5-7 Dec. 1972, USA.

[11] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen, "The ability of object-oriented metrics to predict change-proneness: a meta-analysis", Empirical Software Engineering, Springer, 17 (3), pp. 200-242, June 2012.

[12] L. Marg, L. C. Luri, E. O'Curran, and A. Mallett, "Rating Evaluation Methods through Correlation", 1st Workshop on Automatic and Manual Metrics for Operational Translation Evaluation (MTE'14), Reykjavik, Iceland, 26 May 2014.

[13] R. C. Martin "Agile software development: principles, patterns and practices", Prentice Hall, New Jersey. 2003.

[14] P. Rovegard, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues", Transactions on Software Engineering, IEEE Computer Society, 34 (4), pp. 516–530, April 2008

[15] P. Runeson, M. Host, A. Rainer and B. Regnell, "Case Study Research in Software Engineering: Guidelines and Examples", John Wiley & Sons, 2012.

[16] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis", 35th International Conference on Software Engineering (ICSE 2013), ACM/IEEE Computer Society, pp. 891-900, 18-26 May 2013, USA.

[17] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the Probability of Change in Object-Oriented Systems", Transactions on Software Engineering, IEEE Computer Society, 31 (7), pp. 601-614, July 2005.

[18] S. S. Yau and J. S. Collofello, "Design Stability Measures for Software Maintenance", Transactions on Software Engineering, IEEE Computer Society, 11 (9), September 1981.

[19] Zazworka, N., Seaman, C., Shull, F.: Prioritizing design debt investment opportunities. In: 2nd International Workshop on Managing Technical Debt (MTD' 11), ACM, (2011)

[20] J. Zhang, S. Sagar, an E. Shihab, "The Evolution of Mobile Apps: An Exploratory Study", International Workshop on Software Development Lifecycle for Mobile (DeMobile' 13), ACM, pp. 1-8, Saint Petersburg, Russia, 19 August 2013.