Architectural Implications of Instability Variations in Open-Source Projects

Carlos Carrillo[a], Rafael Capilla[b,*], Victor Salamanca[b], Alejandro Valdezate[b]

[a]*Department of Telematic Engineering and Electronics, Technical University of Madrid, Spain*
[b]*Department of Computer Science, Rey Juan Carlos University, Spain*

Abstract

Architecture instability is one of the consequences of changes in the design during evolution cycles. However, predicting the instability of the architecture seems still challenging as it depends on factors such as cascading effects of wrong design decisions or the frequency of changes of developers that are never updated in the design. Moreover, the granularity of changes in design artifacts may complicate the estimation of instability measures in the architecture. This paper aims to study the evolution of instability in four open-source projects and 69 releases, to suggest ways to predict such instability between releases and inside a release based on changes in the package structure. We also report that the instability variations of sub-releases can be somehow anticipated using the changes of the number of files changed, and we provide results about the instability trends of the package structure accordingly to its complexity and changes in the software architecture.

Keywords: Instability, ripple effect, evolution, design decisions, software architecture, stability, sustainability

1. Introduction

Software architectures evolve over time to cope with changing requirements or handle system changes mainly caused by maintenance operations. In addition, this evolution may cause an architectural mismatch between design and code, leading to architectural drift and erosion when the design diverges from the implementation or when a suboptimal code violates architectural principles [1]. Typically, software architectures are a consequence of design decisions that should be captured and evaluated [2]. In addition, the effect of design decisions on the descriptive architecture (e.g. architectural drift) and on the prescriptive architecture (e.g. architectural erosion) often constitutes an important design problem impacting negatively in software maintenance tasks. Therefore, in many cases, an accumulation of suboptimal design decisions and propagation may have a cascading effect on other parts of the design during evolution cycles ([3], as these decisions impact on the architecture [4]).

Architecture refactoring aims to reduce architecture and design smells [5] [6] which is often estimated using technical debt [7] [8] approaches. In this light, [9] analyzes the modularity and size of modules, separation of concerns, and module dependencies to infer maintainability problems in software architectures. In line with Bouwers, the work from [10] suggests more than 40 metrics that investigate the sustainability of the architecture [11, 12, 13] measuring the dependencies between modules, package dependency cycles, and coupling/cohesion of modules. A refined version described in [14] argues that there is no single metric to estimate software sustainability as this is a complex indicator that should be computed using different estimators like coupling, cohesion, and number of dependencies among others.

Nowadays, one of the principal indicators of architecture erosion is the instability of the architecture or a system. Instability is an indicator that analyzes the frequency of changes of a system over time. According to [15], instability uses the number of dependencies between elements to discover the effect of changes. One estimator of instability is the ripple effect of changes [16], as there is a strong correlation between the ripple effect and the logical stability of software modules based on the cascading impact of changes.

[*]Corresponding author:
Email addresses: `carlos.carrillo@upm.es` (Carlos Carrillo), `rafael.capilla@urjc.es` (Rafael Capilla), `victorsalamanca@gmail.com` (Victor Salamanca), `alejandro@valdezate.net` (Alejandro Valdezate)

The majority of previous studies provide coarse-grained instability estimations and do not study in depth the evolution of the instability inside the releases. Hence, in this work we: (i) study the inter- and intra-instability values, (ii) investigate ways to anticipate or predict instability changes, and (iii) estimate how the evolution of the complexity of the package structure of a project impacts on instability values. Consequently, we advance the state providing a more accurate analysis of instability values and possible correlation between these and the classes and relationships between classes that change. We also investigate the effects of the instability of the files modified over 69, and we analyze how the complexity of the package structure impacts on the instability trend.

The remainder of the paper is organized as follows. In Section 2 we describe the related work about instability metrics. In Section 3 we describe the design method of our study. In Section 4 we report the results of the experimental evaluation and Section 5 discusses the findings. Section 6 describes the threats to the validity of our work. We also discuss the related works in Section 7 and we provide our conclusions and future work in Section 8.

2. Instability metrics

In the following subsections, we detail and compare different approaches to compute the instability in software systems and for different kinds of artifacts. We group them accordingly to the type of element to which an instability formula is applied.

2.1. Instability metrics in packages

Software architecture packages are high-level entities commonly used to describe subsystems or to group related functionality. In many complex systems, some packages depend on others. For instance, the Linux system often requires additional packages when installing and configuring new functionality. As a consequence, a set of dependencies is established between packages and the modifications in one of these packages affect other related packages. Some authors [17] perform a comparison of code querying languages and tools based on an implementation of the instability metric defined by [15], and on the number of classes outside the package that depends on classes inside the package (i.e. AfferentCoupling) and on the number of classes inside the package that depend upon classes outside the package (i.e. EfferentCoupling).

Another work [18] suggests ways to estimate package stability metrics to measure the changes affecting the stability of the architecture and measure the changes that happen during system evolution. The authors estimate the instability of two consecutive releases due to changes in the packages and they provide an aggregate measure of the system instability as the average of the sum of the package instabilities. In addition, the authors in [19] suggest a package stability metric (PSM) based on the changes between package contents and the relationships inside the package. The proposed metric estimates the maintenance effort and computes package stability based on three dimensions: content, internal package connection, and external package connections, as well as eight properties [20] and four types of relationships between classes. More recently, in [21] the authors mention that a package is less stable if it depends on an unstable related package, and they suggest a metric so-called "Degree of Unstable Dependency" as the ratio between the number of dependencies that makes a package unstable (i.e. BadDependency) and the total number of dependencies. Table 1 summarizes the metrics discussed above. Finally, in [19], the authors suggest a new package stability metric based on the changes between package contents and intra- and inter-package connections that validate empirically in five open-source programs. The authors found a negative correlation between the proposed metric and the maintenance effort and a positive correlation between the package stability metrics based on changes in lines of code and class names.

2.2. Instability metrics in components

With respect to the metrics that estimate the instability in software components, we can highlight the following approaches. In [22], the authors propose a new quality model (SQMMA) to compute the stability of a software component using a weighted formula in terms of the number of subclasses, depth of tree hierarchies, the coupling between objects, components invoked, and entry/exit points. The weights for each parameter are assigned using the Analytic Hierarchy Process (AHP) method. In addition, other quality attributes, like modifiability, are computed in terms of stability assigned to a specific weight. The authors compare their results of the stability values with the defect density, so more stable components exhibit less change density.

Author	Instability metric	Instability formula	Instability between versions
[17]	Package Instability	$I = \frac{EfferentCoupling}{AfferentCoupling + EfferentCoupling}$	No
[20]	Property Stability	$Stab_{Property} = \frac{Unchanged_{Property}}{Number_{Property}}$	No
[18]	Aggregate System Instability Change	$ASIC(v) = \frac{\frac{1}{K}\sum_p (I_{v-1} - I_v)_p + \frac{1}{N}\sum_p I_p}{2}$	Yes
[21]	Degree of Unstable Dependency	$DoUD = \frac{BadDependencies}{TotalDependencies}$	No
[19]	Package Stability Metrics	$PSM = \frac{PCS + IPIS + EPIS}{3}$	No

Table 1: Overview of instability metrics for software packages

Works like [23] analyze evolutionary instability metrics between two different versions by measuring the distance between software components. The authors define version stability and branch stability metrics using distances and they compute the structural stability of an artifact for a given period. They also provide an aggregate stability indicator to measure the entire evolution of an artifact.

In [24] the authors suggest a family of instability metrics for measuring the evolution of the architecture instability across multiple releases. They investigate different factors influencing the instability of core components as an indicator of good reusability. To analyze this instability, they propose the so-called Design Instability (DI) metric, aimed to evaluate the changes performed on the architecture between two releases. The proposed formula described in Table 2 includes the components that have changed (c_Comp), added (a_Comp), or removed (r_Comp) in version N+1 with respect to version N and according to the total number of components (n_Comp) in version N. The metric suggests the so-called Calls Instability (CI) index as the changes of the interactions between the software components in release N with respect to release N+1. The summary of these metrics are shown in Table 2.

2.3. Instability metrics in classes

One of the seminal works suggesting the idea of stability in object-oriented (OO) design was [15], who stated a way to measure the instability between OO classes as *"the number of classes inside a particular category that depends upon classes outside this category (i.e. efferent coupling) and divided by the sum of the efferent coupling plus the number of classes outside a category that depend upon classes within this category (i.e. afferent coupling)"*.

The authors in [25] suggest three different instability metrics named System Design Instability (SDI), Class Implementation Instability (CII), and System Implementation Instability (SII), which are used to estimate the evolution of OO systems via the analysis of the instability of object-oriented designs and the classes that have changed. Regarding the SDI metric, the authors conclude that the instability of the project examined is higher in the early stages of the development phase. In addition, in [26], the authors consider a class is stable with respect to a measurement of a previous version if there is no change in that measurement. They provide a formula to estimate the stability of classes applied to a class history, which is computed as the fraction of the number of versions in which a class changes over the total number of versions.

A complementary work described in [28] refines and extends [25] work in order to incorporate the number of inheritances of classes that have changed between two consecutive versions. Another approach [20] describes a

Author	Instability metric	Instability formula	Instability between versions
[22]	Stability	$Stability = -0.19 * Subclasses - 0.21 * Coupling - 0.20 * Hierarchies - 0.18 * EntExt - 0.21 * Communication$	No
[23]	Version Stability	$VS(m) = 1 - \frac{\sum_{i=1}^{p} NCD(S_m, S_n)}{p}$	Yes
[24]	Calls Instability	$CI = \frac{a\_Iter_{N+1} + r\_Iter_{N+1}}{t\_Iter_N + a\_Iter_{N+1} + a\_Iter_{N+1}}$	Yes

Table 2: Instability metric component overview

Author	Instability metric	Instability formula	Instability between versions
[15]	Instability	$I = \frac{Ce}{Ce + Ca}$	No
[25]	Class Implementation Instability	$CII = \frac{LOC_{N+1} - LOC_N}{LOC_N} * 100$	Yes
[26]	Stability	$Stabi_{1..n}(C, M) = \frac{\sum_{i=2}^{n} Stabi_i(C,M)}{n-1}$ $Stabi_i(C, M) = \begin{cases} 1, & M_i(C) - Mi - 1 = 0 \\ 0, & M_i(C) - Mi - 1 \neq 0 \end{cases}$	Yes
[20]	Stability Class	$Stability_{CLASS} = \frac{Stab_{ClassAL} + Stab_{Inteface}}{properties_{CLASS}} + \frac{Stab_{Inhr} + Stab_{Mthd}}{properties_{CLASS}} + \frac{Stab_{Var} + Stab_{varAl}}{properties_{CLASS}} + \frac{Stab_{MthdAL} + Stab_{Body}}{properties_{CLASS}}$	No
[27]	Ripple Effects Measurement	$REM = \frac{NDMC + NOP + NPrA}{NOM + NA}$	No

Table 3: Instability metric class overview

metric to estimate class instability by considering the class relationships and class design. The authors identified class stability factors and they came up with eight class properties used to measure class stability, dividing the unchanged properties by the total number of properties. The authors validated empirically the proposed formula in two industrial client-server systems. The results provided a more accurate estimator of class stability than previous approaches.

Finally, [27] suggested a way to predict class instability in GoF design patterns using a probability formula that relies on an estimation of the ripple effect measure (REM) computed using the number of method calls between classes, the number of methods, and attributes, and the number of polymorphic methods. In this way, the coupling and cohesion between classes participating in instability estimation are major factors to correlate the instability between classes for a given design pattern. The summary of the instability metrics in classes is shown in Table 3.

3. Study Design

We inspired this research in a previous work [29] that describes an instability metric to estimate the ripple effect of design decisions, but we used the formula described by [15] to compute the instability values of open-source projects including the statistical significance of the results. Additionally, we compared our results with those from Alenezi's formula [18] as we discovered certain differences. To advance the state of the art, we investigate in this research how the instability can be estimated more accurately and which is the impact at the architectural level. We conducted a case study [30] [31] in four open-source projects to uncover the estimation and evolution of instability measures, and we came up with the following research questions:

- RQ1. How classes added and removed affect the instability of open-source projects?
  Rationale: As project changes may affect the instability of the overall project, most of these changes are based on adding and removing classes and relationships between classes affecting the overall project structure. Therefore, in this research question we investigate how such changes between project releases modify the instability values.

- RQ2. Can we predict instability variations along the evolution of open-source projects?
  Rationale: In this research question, we attempt to provide trends of the evolution of the instability and use these results to analyze which projects are more stable according to the variations in their instability values. We also attempt to predict future instability changes based on the ratio of modified files.

- RQ3. How the instability of the package structure evolves accordingly to its complexity?
  Rationale: Here we investigate the evolution of the instability based on the increasing complexity of the package structure in open-source projects and we analyze such complexity in the resultant architecture. We focus at the package level as the number of classes in large OSS projects would be unmanageable to visualize the results at the architecture level.

To further investigate the instability, according to the three research questions above, we selected the following open-source projects: Catroid[1], SonarQube[2], dex2jar[3] and Hadoop[4] open-source projects.

We selected these four projects because they are Java projects and the ARCADE tool used in this research requires Java. In addition, the four projects selected have a certain history and contain several versions that show partially their evolution. We used older versions of the projects because they do not use modern development frameworks, so we can have access to the source code of the projects without the need to import third-party libraries. In addition, in the case of Hadoop and SonarQube, we also took into account these projects because of their size and popularity. To apply the instability metrics, we computed the instability of each project at the class level and their dependencies. We also compared our results with one of the formulas described in Section 2 as we found some divergences in the results. Hence, we adopted the following protocol:

[1]https://github.com/Catrobat/Catroid
[2]https://github.com/SonarSource/sonarqube
[3]https://github.com/pxb1988/dex2jar
[4]https://github.com/apache/hadoop

1. Analyze the instability for different releases of the four open-source projects using classes and dependencies and investigate the differences of instability values when classes are added and removed.
2. Provide a way to predict variations in the instability values not only based on the current project structure.
3. Analyze the instability inside the projects and not only between releases in order to provide more accurate instability estimators.
4. Compute the instability of different levels of the different package structure and use these values to estimate how the instability of a certain structure approaches to the instability of a give release.
5. Compare the complexity of the resultant architecture for a given package level with the potential increment of the instability for that release.

4. Data Gathering and Results

In this section, we report the results alongside our observations for each research question addressed in this study.

4.1. Instability changes based on adding and removing elements in open-source project releases (RQ1)

We first answer to **RQ1** reporting our results in the following tables based on the analysis of the instability values for different releases in four open-source projects. We computed such instabilities based on the classes and dependencies added and removed, and we analyzed how such changes affect the instability of the releases, including an analysis of the elements that remain between two consecutive versions. Finally, we provide the ranges of instability variations for the four projects and a statistical analysis to determine possible correlations between the instability measures and their classes and dependencies.

The rationale for the construction of the tables (except for Table 5 which we will explain separately) is as follows. First, we need to provide a column showing the instability of the complete project for each release, as we will need this to evaluate the evolution of the instability for the different releases in RQ2. Moreover, the last two columns showing the number of classes and edges computed using the ARCADE tool are necessary to estimate the instability of the releases. The columns that show the instability of the elements added and removed are necessary to investigate how the instability of a release varies according to the elements added and removed between two consecutive releases. Finally, we include the number of commits and file changes as we will use these numbers to answer RQ2.

In Table 4 we show the instability values for a set of releases of the Catroid project. The third column refers to the instability values of the overall release, which is similar to the second part of the formula suggested by [18]. The fourth and fifth columns show the instability of the classes removed and added between the two releases respectively with respect to the overall number of classes of the previous release. The sixth and seventh columns show the number of commits and files committed mined from GitHub. Finally, the last two columns report the number of classes and edges connecting classes reported using the ARCADE[5] tool. According to the instability ranges defined in our previous work, Catroid exhibits average values of instability for the releases analyzed, excluding the first release in the table (i.e., Catroid 0.5.0a). The biggest difference of instability values between releases is 0.12, but if we exclude the first one, the differences are much lower, ranging around 0.011.

The fourth column shows the instability values of the classes removed between two consecutive releases. We computed such instability taking into account the number of classes and links to the classes removed in a given version with respect to the overall number of classes of the previous version as a way to estimate the impact of the elements that disappear. As we can see, for Catroid, the instability values are rather small, ranging from 0.001 to 0.045. Hence, we computed the average instability of the classes added and removed.

In addition, the fifth column reports the instability values of the packages and classes added between the two versions that we will use later to explain the evolution of the instability across versions. This instability varies between 0.03 to 0.44 for the Catroid releases except for release 0.8.3 which is 0.00, maybe due to the low number of commits and file changes. This result makes a difference with previous works reporting only the instability of the complete

[5]`https://bitbucket.org/joshuaga/arcade`

Table 4: Instability of Catroid releases

Open-source Project	Year	Instability of the complete project	Instability of removed classes and dependencies	Instability of new classes and dependencies	Number of commits	Number of files changed	Number of classes (ARCADE)	Number of edges (ARCADE)
Catroid-v0.5.0a	May-11	0.694819	—	—	—	—	353	2218
Catroid-0.8.0	Jun-13	0.583446	0.019218	0.443996	5844	1472	1273	9483
Catroid-0.8.3	Jul-13	0.574862	0.045065	0.000000	80	84	1185	8940
Catroid-0.9.0	Aug-13	0.583432	0.001093	0.038825	398	551	1280	9910
Catroid-0.9.10	Aug-14	0.572358	0.002431	0.067633	671	1248	1428	10837

releases and not for the elements added. Please note that the differences of instability between versions for the classes added or removed are computed in absolute values.

If we pick randomly two consecutive releases from Table 4 (e.g. Catroid 0.8.3 and Catroid 0.9.0) and we subtract from release 0.9.0 the instability of the classes removed from release 0.8.3 and we sum the instability of the new classes added, the result is: 0.574862 -0.001093 +0.038825 = 0.612594, which is different from the instability computed for version 0.9.0 (i.e. 0.583432). This means that it is not possible to infer the instability of a given version should be equal to the instability of the previous release subtracting the instability of the classes removed in the previous version plus adding the instability of the new classes. The main reason is that the reorganization of the links when adding and removing classes may lead to variations in the instability values accordingly to a different topology.

However, if we take a look at the numbers in Table 5, we have included two new columns with respect to Table 4 showing the elements (i.e., classes and their relationships) that remain between two consecutive releases. In column 5 of Table 5 we show the instability of those common elements referred to previous releases, while in column 6 we show the same value for the current release.

Table 5: Instability of Catroid releases including the elements that remain between versions

Open-source Project	Year	Instability of the complete project	Instability of removed classes and dependencies	Instability of the elements that remain between two releases referred to the previous version	Instability of the elements that remain between two releases referred to the current version	Instability of new classes and dependencies
Catroid-v0.5.0a	May-11	0.694819	—	—	—	—
Catroid-0.8.0	Jun-13	0.583446	0.019218	0.675601	0.139450	0.443996
Catroid-0.8.3	Jul-13	0.574862	0.045065	0.538381	0.574862	0.000000
Catroid-0.9.0	Aug-13	0.583432	0.001093	0.573769	0.544607	0.038825
Catroid-0.9.10	Aug-14	0.572358	0.002431	0.581001	0.504725	0.067633

If we look to the colored numbers referring to the instability values of Catroid 0.8.3 and 0.9.0, the numbers in orange show that the instability of the full project for release 0.8.3 is 0.574862 as the sum of the instability of the classes removed in that version (i.e. 0.001093) plus the instability of the common elements that remain in version 0.9.0 (i.e. 0.573769). Similar reasoning applies for Catroid 0.9.0, where its instability is shown in magenta color (i.e., 0.583432) as the sum of the instability of the elements that are common to both releases plus the instability of the new elements added for that release (i.e. 0.038825). This means that the instability computed for each release can be computed using this formula, but according to the results of Table 4, the instability of a given release is not the sum of the instability of the previous release plus the instability of the new elements minus the instability of the elements removed, as the shape of the graph relating classes and links can change. Only if taking into account the common elements between versions, we can double-check the instability values are computed correctly. We also need to remark that the instabilities of the common classes are slightly different as the value for versions 0.8.3 and 0.9.0 is computed with respect to the total number of classes and elements for each version.

In the following tables, we show the instability results for SonarQube, dex2jar, and Hadoop projects. In the case of SonarQube (Table 6), we observe higher values of instability values than in Catroid and closer to the limits of high instability range. Furthermore, the instability of the new classes added between versions exhibits small values ranging between 0 and 0.02. Two of the releases show that the instability of the classes added is equal to 0 and caused by the same reason regarding a few number of elements added between versions.

Table 6: Instability of SonarQube releases

Open-source Project	Year	Instability of the complete project	Instability of removed classes and dependencies	Instability of new classes and dependencies	Number of commits	Number of files changed	Number of classes (ARCADE)	Number of edges (ARCADE)
SonarQube-5.0.1	Feb-15	0.671386	—	—	—	—	1264	8893
SonarQube-5.2	Nov-15	0.662149	0.003629	0.021131	4033	10068	1316	9532
SonarQube-5.3	Jan-16	0.661456	0.000000	0.000000	561	2382	1317	9354
SonarQube-5.5	May-16	0.677134	0.019905	0.011001	1163	8299	1301	9143
SonarQube-6.0	Aug-16	0.676201	0.000000	0.000000	856	4074	1303	9168

The last two tables belong to the instability values computed for dex2jar (Table 7) and Hadoop (Table 8) projects. While DexJar has a bit lower instability values than SonarQube, Hadoop shows similar values to SonarQube, so both are almost close to the limit between average and high instability ranges. In the case of dex2jar, we did not find instability values equal to zero in the classes added, which vary between 0.006 and 0.033. In Hadoop, only one release shows zero value of instability and the classes added exhibit an instability ranging from 0.01 to 0.32, which is a bit higher than for SonarQube and dex2jar. The main reason for this relies on the bigger size and higher complexity of the Hadoop project.

Table 7: Instability of dex2jar releases

Open-source Project	Year	Instability of the complete project	Instability of removed classes and dependencies	Instability of new classes and dependencies	Number of commits	Number of files changed	Number of classes (ARCADE)	Number of edges (ARCADE)
dex2jar-0.0.9.5	Jan-12	0.652549	—	—	—	—	1122	10733
dex2jar-0.0.9.7	Jan-12	0.632887	0.097456	0.015328	27	180	1007	9839
dex2jar-0.0.9.10	Oct-12	0.637224	0.004343	0.033145	65	92	1038	10259
dex2jar-0.0.9.12	Dec-12	0.637418	0.001022	0.003118	31	73	1043	10339
dex2jar-0.0.9.15	Jun-13	0.633606	0.019646	0.006282	37	257	1028	10199

Table 8: Instability of Hadoop releases

Open-source Project	Year	Instability of the complete project	Instability of removed classes and dependencies	Instability of new classes and dependencies	Number of commits	Number of files changed	Number of classes (ARCADE)	Number of edges (ARCADE)
Hadoop-0.10.1	Jan-07	0.675264	—	—	—	—	1697	15776
Hadoop-0.12.0	Mar-07	0.678582	0.185532	0.019068	159	355	1394	12147
Hadoop-0.13.0	Jun-07	0.678582	0.000000	0.000000	184	502	1394	12147
Hadoop-0.18.0	Jun-09	0.674294	0.001153	0.151135	1633	1869	1663	14030
Hadoop-0.19.0	Jun-09	0.674504	0.001189	0.118946	571	2256	1960	18007
Hadoop-0.20.0	Jun-09	0.684535	0.514291	0.006231	558	2085	3363	38295

Finally, we report in Figure 1 a comparison of the lower and upper values of instability for the classes and their dependencies added and removed between versions. As we can observe, Catroid and Hadoop projects show the biggest magnitudes of instability values for the elements added. In the case of elements removed, the overall instability values vary in a small range for all projects and only Hadoop exhibits bigger magnitudes. These numbers could be used as indicators of variation of instability along with the evolution of the releases and use these to identify which projects vary more often. We observed that when the instability varies in smaller ranges, the projects tend to be more stable. The complete list of instabilities for the releases examined can be found at [6].

[6] `https://github.com/CCS-repository-public/Implications-OSS-Projects/blob/main/Implications-OSS-Projects.`

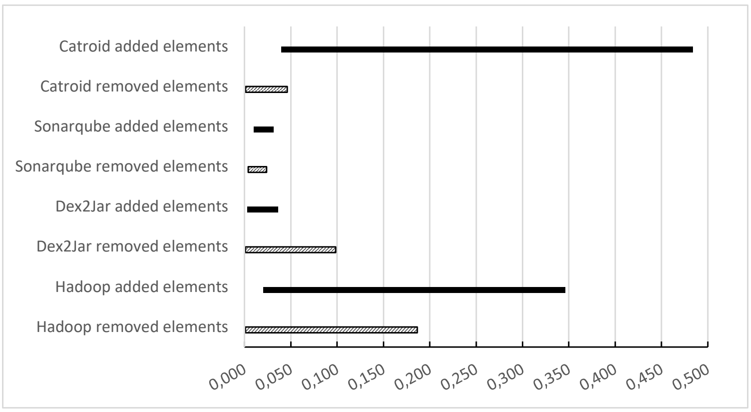


Figure 1: Ranges of instability values of the classes and dependencies added and removed. Instability ranges of the elements added are shown in solid black lines and the instability ranges of the removed elements are shown in grey color lines.

Summary: The tables shown above report instability values for different releases of four open-source projects. The instability values based on Martin's formula [15] for the four projects exhibit average values of stability, but SonarQube and Hadoop projects are at the limits of the range where the stability of these projects can be considered to start decaying. When we compared the instability of the projects analyzed with the formulas described in Table 1, we derived the following findings. The formula from [21] and [19] estimate the instability values in a different way than ours and focused on architectural smells. Moreover, it is difficult to derive a common pattern of the differences of the instability in the projects investigated, apart from that these exhibit small values, such as the case, between Catroid and Hadoop. Although Hadoop is much more complex, the higher instability of the classes added in Catroid for the releases analyzed is 0.44 while Hadoop is 0.32. This leads us to think that the size of the projects is not the main factor as a predictor of its instability, but rather the number of elements and relationships of the entities involved in the project changes.

Statistical analysis: In order to discover the possible correlation between the variables of the columns of the tables above, we run a Pearson test using the Statgraphics tool [7]. We investigated the correlation between the instability (i.e. dependent variable) and the number of classes and edges of a release as independent variables. Hence, we defined the following hypotheses:
H0: There is no significant association between the instability and the classes and edges of a project release.
H1: There is a significant association between the instability and the classes and edges of a project release.

We show the results of the Pearson test and the p-value in Table 9. We found a positive correlation between the instability and the classes and dependencies in half of the projects (i.e. dex2jar and Hadoop) as the p-value is lower than 0.05. Hence, we can reject the null hypothesis H0 for these two projects, while for Catroid ans Sonarqube we couldn't find such correlation. Those values that exhibit a linear correlation in Hadoop are shown in Figure 2.

Table 9: Pearson Correlation between instability values and classes and relationships

Instability of the complete project	Number of classes		Number of edges	
	Pearson correlation	p-value	Pearson correlation	p-value
Catroid	-0.1317	0.6144	0.0206	0.9374
SonarQube	-0.0578	0.857	0.04	0.8787
dex2jar	-0.8179	0	-0.847	0
Hadoop	-0.5787	0.0149	-0.5828	0.0141

pdf
[7]https://www.statgraphics.com/download18

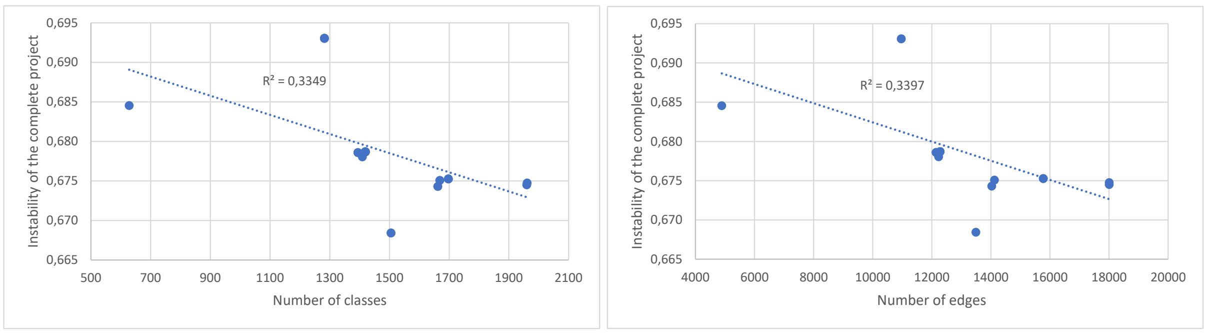


Figure 2: Linear regression of the instability values for the number of classes (left figure) and edges (right figure) belonging to the Hadoop project

In addition, as we did not find a linear correlation between the instability and the classes and dependencies for Catroid and Sonarqube projects, we investigated the possibility of a non-linear correlation using the Statgraphics tool. As a result, we found a polynomial regression between the instability and the number of classes and edges for those projects. In Figure 3 we show the results of this kind of correlation for Catroid (Figure 3 left) and Sonarqube (Figure 3 right) projects. In both cases, the p-value is less than 0.05 (i.e. 0.0002 for Catroid and 0.0167 for Sonarqube), which means that we can reject the null hypothesis H0 and confirm that both variables are correlated.

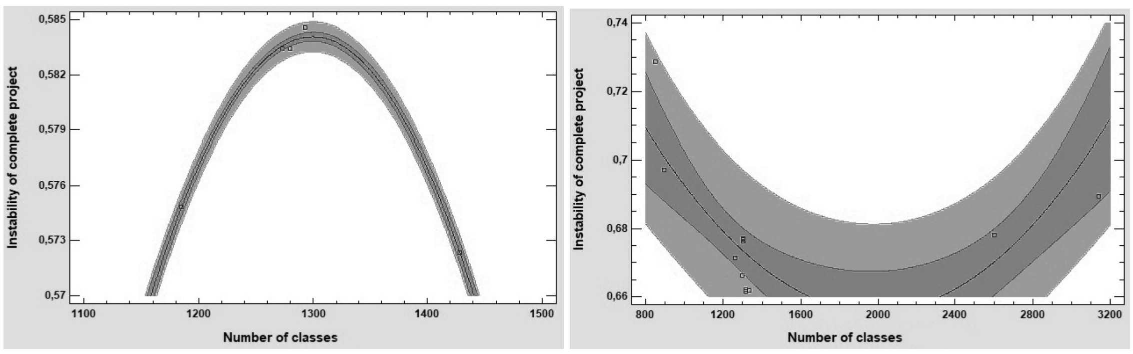


Figure 3: Polynomial regression model between the instabilities and the classes recovered with ARCADE for Catroid (left figure) and Sonarqube (right figure) projects

4.2. *Instability across evolution (RQ2)*

In order to answer to **RQ2**, we investigated the evolution of the instability in the four open-source projects and the evolution intra-releases. In the first case, the comparison of the instability in the four projects analyzed is shown in Figure 4. Each project shows not only how the instability evolves for the versions analyzed but also a dotted line highlights the trend of such instability. For Catroid and dex2jar, both projects show a similar instability pattern and the inclination of the trend is higher than in SonarQube and Hadoop which exhibit a more flat instability trend. All instability values are shown in the same range to be easily compared.

In the case of SonarQube, the instability tend to decrease first and after to increase its instability in the last two versions, which exhibits an opposite trend to Catroid and dex2jar. One reason could be that SonarQube is a very active tool and project where the existing functionality is being modified often. However, according to Figure 1, the number of features added is higher than in other projects which leads to higher instability values (i.e. above 0.66) similar to Hadoop releases. In the case of dex2jar, the instability decreasing to become more stable in the last versions, while Hadoop looks different as it exhibits a flat figure of the instability values with a small increment in the last release examined.

Additionally, we analyzed the instability of the four projects at lower granularity levels, such as we show in the four tables below. Regarding the Catroid project, we observed the instability of ten sub-releases between releases

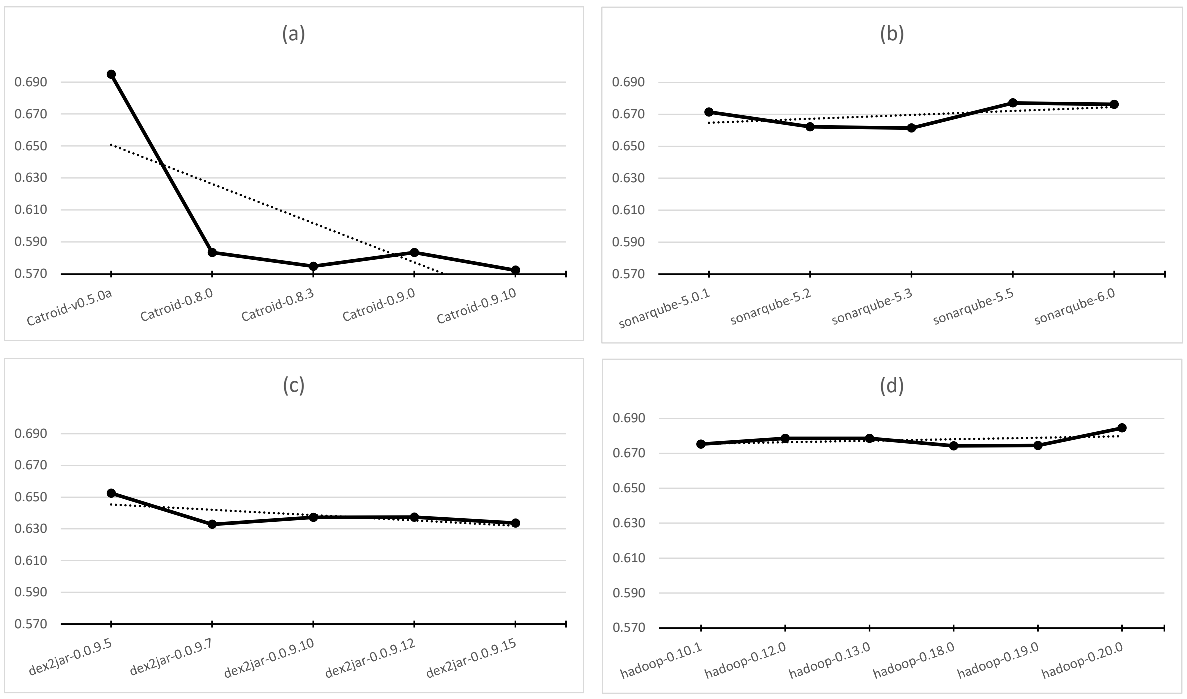


Figure 4: Evolution and trend of the instability for (a) Catroid, (b) SonarQube, (c) dex2jar and (d) Hadoop projects. The dotted line in each project represents the trend of the instability values. The solid line refers to the instability values of Tables....CARLOS.

0.9.0 and 0.9.10, but to avoid an excessive number of data, we grouped the data for pairs of consecutive releases (e.g. 0.9.1/2, 0.9.3/4, and so on), such as Table 10 shows. As we can observe, there is no change in the instability values until release 0.9.5 while the instability values of the elements added and removed are 0.00 in the first two pairs of releases. A similar thing happens when we jump from release 6 to releases 7 and 8. Finally, the last two pairs of releases show a slight change of the overall instability values (i.e. a certain decrease) mainly cause by a different structure of the classes and dependencies for the new elements added.

Table 10: Instabilities of Catroid sub-releases

Versions of Catroid	Number of modified files	Number of commits	Instability of the complete project	Ratio Modified files / Total files	Instability of removed elements	Instability of new elements
v-0.9.1/0.9.2	208	29	0.5834	0.1142	0.000000	0.000000
v-0.9.3/0.9.4	214	184	0.5834	0.1175	0.000000	0.000000
v-0.9.5/0.9.6	671	155	0.5846	0.3683	0.002431	0.011141
v-0.9.7/0.9.8	384	157	0.5846	0.2108	0.000000	0.000000
v-0.9.9/0.9.10	345	146	0.5724	0.1894	0.000000	0.057827

We did the same for the other three projects such as shown in Tables 11, 12, and 13. Although the identification of the causes of such instability changes could be uncertain, we discuss below the future trends of instability based on the ratio of modified files, independently of the roots of such changes, and hence, provide some guidance to developers about how stable or unstable a project could be.

In Figure 5 we depict the graphical view of the values shown in previous tables and for each of the four projects. The solid line in each sub-figure represents the evolution of the instability for the five pairs of sub-releases analyzed while the dotted lines represent the ratio of the modified files for each sub-release regarding the total number of modified files according to ecuation (2). Each project behaves differently as we explain below.

In the case of Catroid (sub-figure 5(a)) , while the first four releases exhibit the same instability, there is a small increment for releases 5/6 and 7/8. In the last two releases, the instability significantly drops with respect to the previous sub-releases but still varying in a small range of values around 0.012. Hence, the peak value shown for the

Table 11: Instabilities of SonarQube sub-releases

Versions of SonarQube	Number of modified files	Number of commits	Instability of the complete project	Ratio Modified files / Total files	Instability of removed elements	Instability of new elements
v-4.5.7/5.0.1	6629	2296	0.6714	0.1918	0.450541	0.049623
v-5.1.2/5.2	10068	4033	0.6621	0.2913	0.003629	0.021131
v-5.3/5.4	6586	999	0.6618	0.1905	0.000891	0.004957
v-5.5/5.6.7	5397	1185	0.6762	0.1561	0.020844	0.007744
v-6.0/6.1	5888	1034	0.6769	0.1703	0.000990	0.000000

Table 12: Instabilities of dex2jar sub-releases

Versions of dex2jar	Number of modified files	Number of commits	Instability of the complete project	Ratio Modified files / Total files	Instability of removed elements	Instability of new elements
v-0.0.9.6/0.0.9.7	180	27	0.6329	0.2821	0.097456	0.015328
v-0.0.9.8/0.0.9.9	61	39	0.6358	0.0956	0.000000	0.022613
v-0.0.9.10/0.0.9.11	73	41	0.6373	0.1144	0.005242	0.011833
v-0.0.9.12/0.0.9.13	263	33	0.6369	0.4122	0.001175	0.004568
v-0.0.9.14/0.0.9.15	61	20	0.6336	0.0956	0.018430	0.003735

Table 13: Instabilities of Hadoop sub-releases

Versions of Hadoop	Number of modified files	Number of commits	Instability of the complete project	Ratio Modified files / Total files	Instability of removed elements	Instability of new elements
v-0.11/0.12	355	159	0.678582	0.0616	0.185532	0.019068
v-0.13/0.14	965	419	0.678041	0.1674	0.000000	0.010242
v-0.15/0.16	1178	756	0.678687	0.2043	0.000000	0.005233
v-0.17/0.18	1625	683	0.674294	0.2818	0.001127	0.138945
v-0.19/0.20	1643	993	0.684535	0.2849	0.619837	0.494089

ratio of modified files anticipates a small change in the instability of the next releases.

For SonarQube, (Figure 5(b)) the project seems more dynamic than Catroid and there is a peak in the ratio of the modified files that anticipates a change in the instability in the next four sub-releases as anticipation of future changes.

Regarding dex2jar, we have two main peaks in the ratio of the modified files. The first ratio drops significantly (from 0.2821 to 0.0956) and the peak increases from 0.1144 to 0.4122. However, this project seems to be an outlier as based on the numbers and the flat figure of the instability values we cannot provide additional conclusions based on the ratio of modified files. The second peak shows an increment in the ratio of modified files going from 0.1144 to 0.4122, but the instability trend remains almost stable.

Finally, in the case of Hadoop, only the last pair of releases increase its instability as the result of an accumulation of a growing ratio of modified files. Hence, as bigger is the ratio of modified files it is more likely the instability varies more often.

In some cases, we observed the values of the dotted lines are aligned future changes in the instability values when the ratio of committed files grow, but in other projects the solid line is not influenced by the dotted line, maybe because uncertain requirements.

$$RatioOfModifiedFiles_i = \frac{NumberOfModifiedFiles_i}{\sum_{i=1}^{N} NumberOfModifiedFiles_i} \quad (1)$$

Finally, we manually checked the presence of elements added or removed in the releases where the instability of the elements added or removed is equal to 0, so we followed the steps given below:

1. We downloaded from Github the pair of releases we wanted to investigate when new elements were added or removed.

2. We expanded the java files to inspect the differences between both versions provided by the Diff command.

3. We sought manually for the word “class” to find new classes added or removed as these are shown in different colors. We did not investigate the dependencies between classes as these are hard to identify by a manual

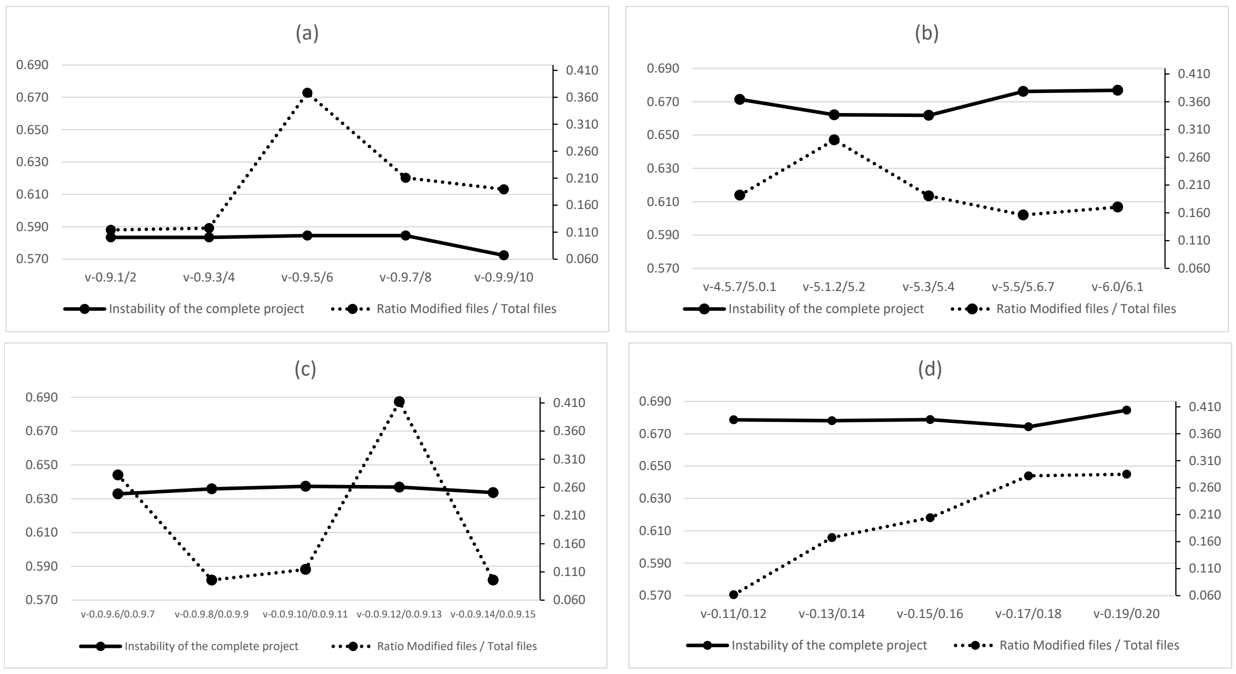


Figure 5: Comparison between Instability and ratio of the modified files for (a) Catroid, (b) SonarQube, (c) dex2jar and (d) Hadoop

inspection inside the details of each class.

4. We classified as new classes those shown in green color as indicated by Github for the new SLOC.

5. The classes removed are shown in orange color but we are unable to identify a class removed if the file containing such class has been deleted.

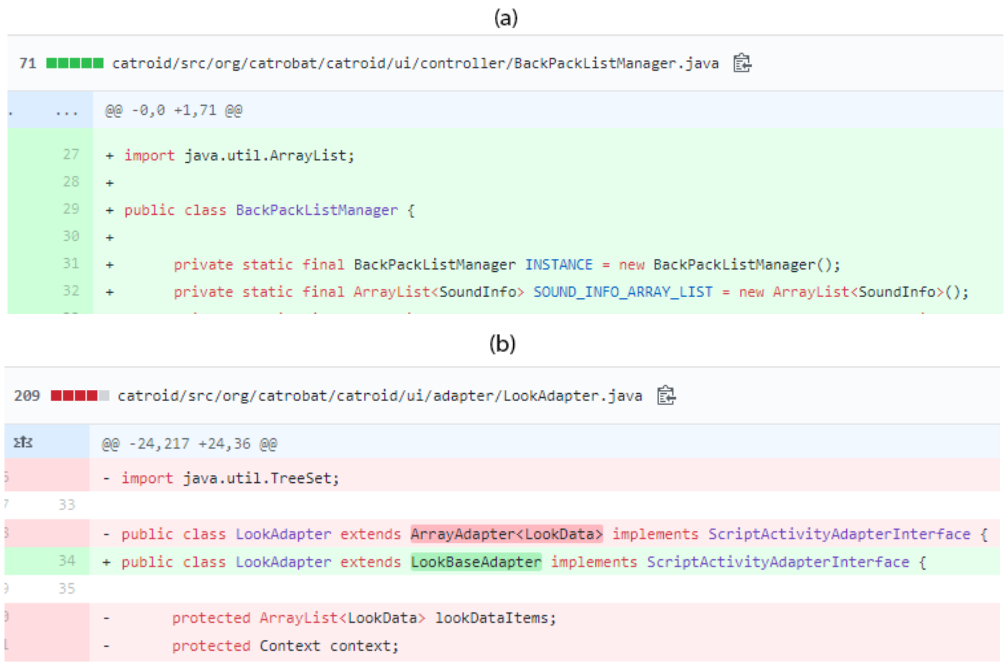
For Catroid releases 0.9.1/2, we did not find elements added or removed, so the instability values for those elements seem correct. However, for releases 0.9.3/4 we found 15 classes added and 4 removed, but the instability of those elements is so small that becomes irrelevant and out of range for the values shown in Table 10. In Figure 6 we can see an excerpt of classes added (i.e., shown in green color) and classes removed (i.e., shown in light orange color). As we can observe in Figure 6(a), class `BaclPackListManager` was added, while in Figure 6(b) class `LookAdpater` was first removed and after added again, maybe because a change in the details of the class or a change in a dependency to another class.

Therefore, even if we can automate the inspection of the differences between two Catroid releases and detect the dependencies added or removed, it seems clear that once the instability computed for the elements added and removed is equal to 0, the appearance of elements found by manual inspection is irrelevant for the overall instability values of the project and it does not have a visible impact on the analysis of the evolution of the instability values. Consequently, we can accept as valid the instability values shown in Table 10.

Summary: Predicting the instability of open-source projects between releases when the values vary in a small range seems challenging. Therefore, as adding new elements to the formulas investigated in the related works did not contribute too much to provide additional insight, as we only found that examining the ratio of the files modified could be used for predicting changes in the future instability values. Also, in those cases where the instability of the elements added or removed is 0, the only way to predict changes is to examine manually the presence of elements added or removed.

4.3. Evolution of the instability of the package structure (RQ3)

In this section, we address **RQ3** regarding the connection between the instability and the structure of open-source projects. Here we try to estimate how the instability evolves accordingly to the increasing complexity of the package


(a)

```
71 catroid/src/org/catrobat/catroid/ui/controller/BackPackListManager.java
@@ -0,0 +1,71 @@
+ import java.util.ArrayList;
+
+ public class BackPackListManager {
+
+       private static final BackPackListManager INSTANCE = new BackPackListManager();
+       private static final ArrayList<SoundInfo> SOUND_INFO_ARRAY_LIST = new ArrayList<SoundInfo>();
```

(b)

```
209 catroid/src/org/catrobat/catroid/ui/adapter/LookAdapter.java
@@ -24,217 +24,36 @@
- import java.util.TreeSet;

- public class LookAdapter extends ArrayAdapter<LookData> implements ScriptActivityAdapterInterface {
+ public class LookAdapter extends LookBaseAdapter implements ScriptActivityAdapterInterface {

-       protected ArrayList<LookData> lookDataItems;
-       protected Context context;
```

Figure 6: Excerpt of classes add and removed in Catroid releases 0.9.3 and 0.9.4

structure changes and compare the architectures of two different versions in order to evaluate the impact of changes in the design. To do so, we adopted the following protocol.

1. Select one release for each of the open-source projects analyzed.
2. Compute the instability for each package level without classes.
3. Compare how close the instability values for each package level is with respect to the instability of the overall project release.
4. Continue with the next package level of the project structure until we approach the closest instability value to the project release.
5. In case the instability value of a given package level is higher than the instability of the release, we compute the instability for that level including the classes of the packages (excluding classes of third-party code) and we compare the instability values again until we approach the instability of the overall project.

As a result, we computed the instability values for each package level and for the open-source projects analyzed. In the first case, we show these values for the Catroid project release 0.8.3 shown in Table 14. As we can observe in the table, for each of the levels (Level 1 is the lowest) we describe in the fifth column the package structure for one of the Catroid releases. The second and third columns show the number of packages for that level of the project with and without external libraries, respectively[8]. In addition, the fourth column of Table 14 shows the instability values for the number of nodes belonging to the third column until we reached the package structure Level 7. This means there are no more sublevels of packages for this project.

Finally, we can compare the instability of the Catroid project for that release (i.e., 0.574862) with the instability of the last package level (i.e., (0.566484). The difference between the two instability values is 0.0084. In this research,

[8]The reason is because Java and other external libraries are often included in the structure the project release and in order to provide more accurate instability estimators we consider the instability values without the external libraries created by others.

Table 14: Instability of Catroid-0.8.3 packages

Project Release	Packages	Packages included	Instability value	Package structure
Catroid-0.8.3	1185	915	0.574862	com.badlogic.gdx.graphics.g3d.loaders.g3d.chunks.ChunkReader
Level 7	183	149	0.566484	com.badlogic.gdx.graphics.g3d.loaders.g3d.chunks
Level 6	180	146	0.561104	com.badlogic.gdx.graphics.g3d.loaders.g3d
Level 5	162	128	0.533906	com.badlogic.gdx.graphics.g3d.loaders
Level 4	128	94	0.430124	com.badlogic.gdx.graphics.g3d
Level 3	97	63	0.278613	com.badlogic.gdx.graphics
Level 2	69	39	0.092363	com.badlogic.gdx
Level 1	46	30	0.119192	com.badlogic

we did not establish a specific threshold value to consider the difference of instability significantly acceptable to decide the include classes to approach the final instability of the architecture computed for that Catroid release. However, what is more interesting is to observe the trend by which the instability values increase on each level as more design decisions are made to include more packages. As the difference of instability values between the package Level 7 and the instability of the architecture for that project seems small, we can conclude that the instability of Level 7 can be used as an estimator of the instability for that release and its architecture as well without computing the instability of the classes of packages.

In the case of SonarQube and according to Table 15 we show the instability values for six package levels belonging to release 6.0. In this case, the instability started increasing up to level 5 which approaches the instability value of the project (i.e., 0.676201). However, the instability at Level 6 (i.e., 0.705478) beats the instability of the project release. In this case, we decided to compute again the instability of Level 6 but including the classes of those packages and excluding the Java classes of external libraries (i.e., as these classes should not be maintained by SonarQube developers). As a result, the instability for Level 6 including the classes is 0.675302. The difference between this instability of the release and the instability of the new Level 6 is approximately 0.0009, which is smaller than in the previous case. Consequently, in some cases, we can compute more accurately the instability of a release and its architecture including the classes of the project for a given package level.

Table 15: Instability of SonarQube 6.0 packages

Project Release	Packages	Packages included	Instability value	Package structure
Sonar 6.0	1303	932	0.676201	ch.qos.logback.core.joran.event.stax.StaxEvent
Level 6	219	177	0.705478	ch.qos.logback.core.joran.event.stax
Level 5	204	162	0.685785	ch.qos.logback.core.joran.event
Level 4	144	102	0.604672	ch.qos.logback.classic.db
Level 3	75	33	0.467874	ch.qos.logback.classic
Level 2	56	18	0.374336	ch.qos.logback
Level 1	30	10	0.337381	ch.qos
	259	217	0.675302	Level 6 including the classes

In our third and fourth projects, we observed a similar case as in SonarQube. The instability values of Hadoop and dex2jar packages increase and overcome the instability of their releases when Hadoop and dex2jar reach Level 4. Like in SonarQube, we computed the instability for the project shown in Tables 16 and 17.

The difference in dex2jar between the instability of the release (i.e., 0.633606) and the instability of Level 5 including their classes (i.e. 0.613404) is around 0.02, which is smaller than in Hadoop but not so small like in Catroid and SonarQube. Hence, computing the instability for package Level 5 including their classes, that it is shown in the last row of Table 16 we approach the instability of the overall project for that release and use this technique to estimate the instability of the project more accurately as the complexity of the package structure increases from one package level to the next one.

The same happens with the Hadoop project where the instability of the release (i.e., 0.674584) is lower than the instability at Level 4 (i.e., 0.700761 17), so if we include the classes for that level the instability decreases to 0.5991897.

Table 16: Instability of dex2jar-0.0.9.15 packages

Project Release	Packages	Packages included	Instability value	Package structure
dex2jar-0.0.9.15	1028	839	0.633606	com.android.dx.dex.code.form.Form10t
Level 5	163	130	0.672091	com.android.dx.dex.code.form
Level 4	142	109	0.636221	com.android.dx.dex.code
Level 3	102	69	0.599056	com.android.dx.dex
Level 2	68	37	0.590601	com.android.dx
Level 1	36	24	0.537687	com.android
	283	250	0.613404	Level 5 including the classes

Table 17: Instability of Hadoop 0.18.0 packages

Project Release	Packages	Packages included	Instability value	Package structure
Hadoop 0.18.0	1663	1122	0.674584	org.jets3t.service.impl.soap.axis._2006_03_01.Grant
Level 6	252	203	0.730687	org.jets3t.service.impl.soap.axis._2006_03_01
Level 5	245	196	0.725030	com.sun.net.ssl.internal.ssl
Level 4	208	159	0.700761	com.sun.net.ssl.internal
Level 3	138	89	0.648987	com.sun.net.ssl
Level 2	84	37	0.649632	com.sun.net
Level 1	37	15	0.625641	com.sun
	813	764	0.5991897	Level 4 including the classes

Moreover, in Figure 7 we can observe for the analyzed SonarQube release, the architecture belonging to the package structure of Level 2. To avoid excessive complexity in the figure, we do not represent the packages and relationships belonging to Java libraries, as we are more interested only in the instability of the packages and classes maintained by SonarQube developers. The packages shown belong to the `org.sonar` project. In Figure 7 we have 13 Sonar packages and 10 dependencies, and the instability value for those packages is 0.244318, which is a bit lower than the instability shown in Table 15 because we did not include the Java packages.

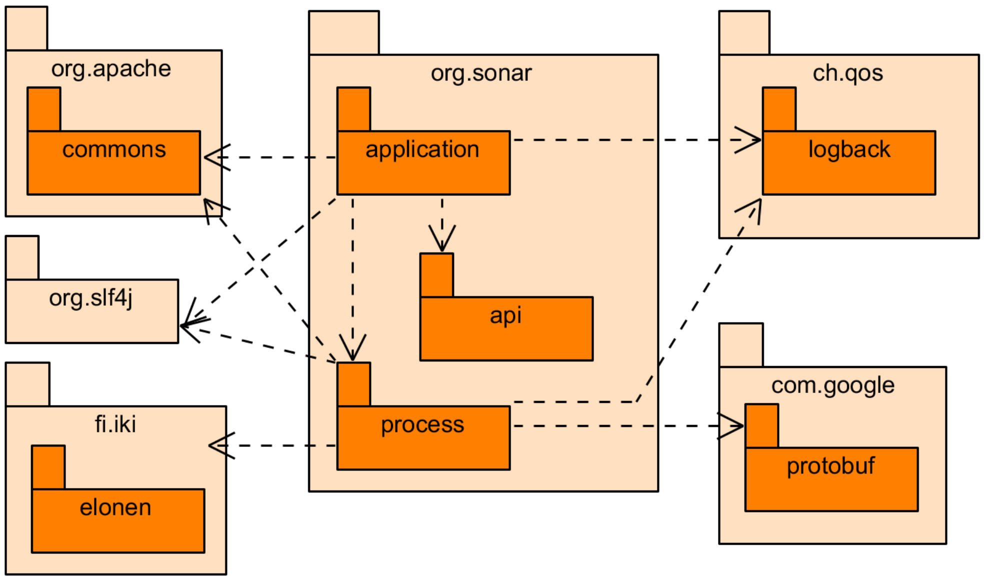


Figure 7: SonarQube packages of level 2 without including Java libraries

In Figure 8 we can see the packages and dependencies for Level 3 so the numbers increased up to 24 packages and 35 dependencies while the instability value grew up to 0.448775 (i.e. again a bit lower than the value shown in Table 15 as we did not include the invocation of Java packages). The evolution of the architecture of the SonarQube project from Level 2 to Level 3 shows the significant increment in the number of classes and dependencies between them which justifies the instability increases accordingly to the complexity of the package structure.

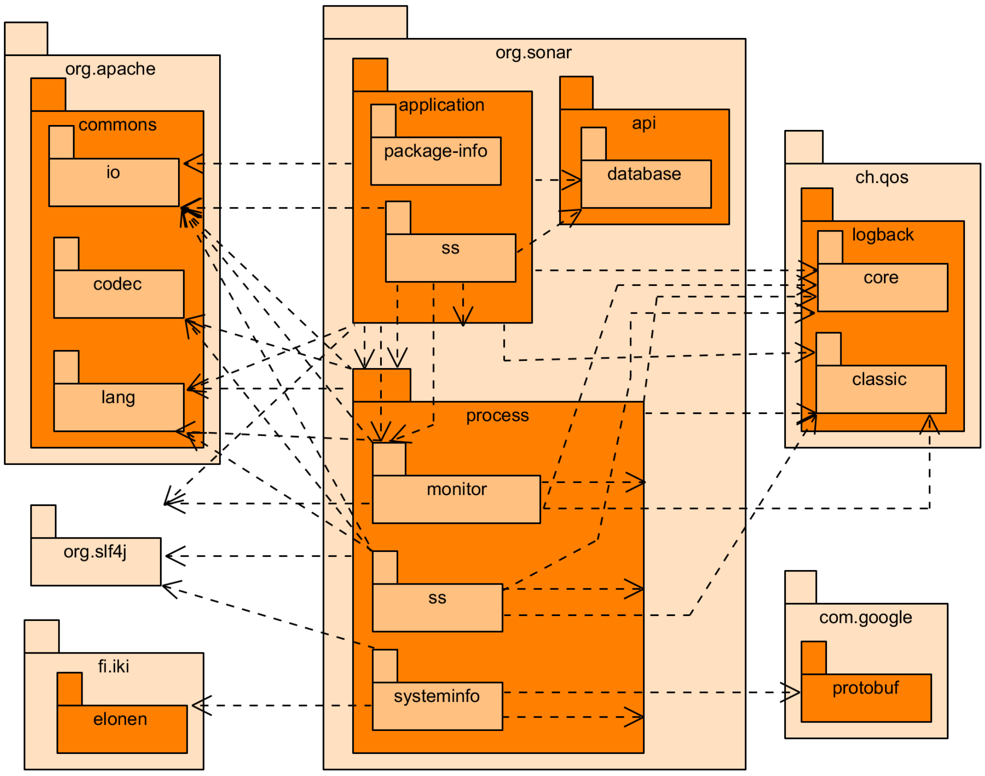


Figure 8: SonarQube packages of level 3 without including Java libraries

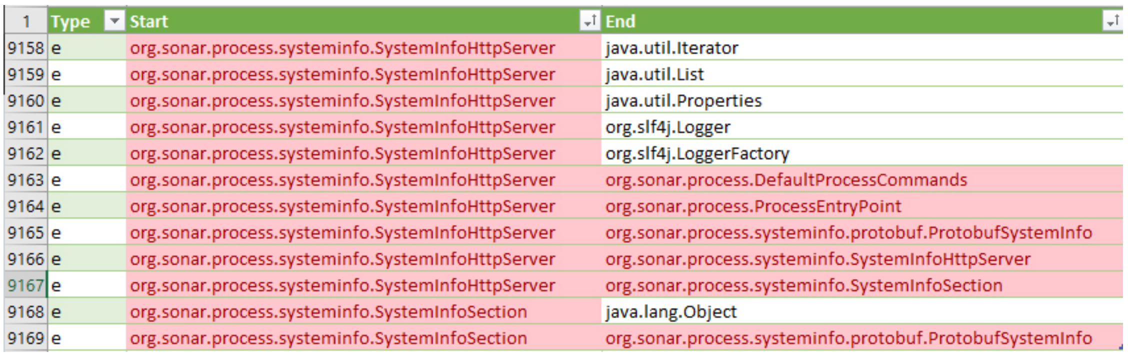
Finally, Figure 9 shows an excerpt of the dependencies between SonarQube packages distilled from the long list of dependencies recovered using the ARCADE tool. The direction of the dependencies goes from the Start column to the End column. The figure displays some dependencies from the package `org.sonar.process.systeminfo` to the `Java.util.*` library but also to some other Sonar packages and third-party code such as `org.slf4.*`. Hence, this is the way we followed to filter out the dependencies of the packages for each package level such as shown in Figures 7 and 8.

Summary: The changes in the package structure in open-source projects show how the instability evolves without the need to compute the instability of dozens of classes, even if the complexity of the architecture seems more complex. However, such instability values tend to grow until a concrete level of depth in the package structure and after tends to stabilize or in some cases it could decrease. At that points, it seems unnecessary to analyze packages at lower levels of depth. Consequently, we observe a positive correlation between the package structure and the trend of the instability values for a given project.

5. Discussion

Based on our results, we provide in this section the discussion of our results.

Comparing our results with previous studies, only the formula from [18] provide similar results to ours, but when we simulated Alenezi's formula for the first two versions of the PDFBbox project, the instability values we obtained



1	Type	Start	End
9158	e	org.sonar.process.systeminfo.SystemInfoHttpServer	java.util.Iterator
9159	e	org.sonar.process.systeminfo.SystemInfoHttpServer	java.util.List
9160	e	org.sonar.process.systeminfo.SystemInfoHttpServer	java.util.Properties
9161	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.slf4j.Logger
9162	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.slf4j.LoggerFactory
9163	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.sonar.process.DefaultProcessCommands
9164	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.sonar.process.ProcessEntryPoint
9165	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.sonar.process.systeminfo.protobuf.ProtobufSystemInfo
9166	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.sonar.process.systeminfo.SystemInfoHttpServer
9167	e	org.sonar.process.systeminfo.SystemInfoHttpServer	org.sonar.process.systeminfo.SystemInfoSection
9168	e	org.sonar.process.systeminfo.SystemInfoSection	java.lang.Object
9169	e	org.sonar.process.systeminfo.SystemInfoSection	org.sonar.process.systeminfo.protobuf.ProtobufSystemInfo

Figure 9: Type of edges in excel file

in versions 1.5 and 1.6 were 0.593 in both cases, so similar to 0.568 instability value provided by [18]. However, in the formula proposed by Alenezi et al., the authors divide the result by 2, which does not make sense as the values obtained would be half. As we did not find much information about how they obtained the packages and which are the dependencies between them, it is hard to know if their results are valid.

Regarding our case study, we observed that the instability of a given release is not exactly the same as in the previous version taking into account the instability of the elements added and removed. Only considering the common elements between releases, we can double-check that the instability values for a given release are computed correctly. Consequently, the reorganization of packages and dependencies may lead to a different topology and possibly variations in the instability values. In this way, software architects can slightly reduce the instability if they modify the packages and classes structure of the project.

The proposed formula to predict the intra-instability of a project using the ratio of the modified files seems to anticipate a change in the next sub-release when the ratio of modified files varies above the average value, but we cannot confirm the magnitude of the change in the future instability values. Therefore, by observing these hot-spots in the releases, we can anticipate to possible variations in the instability values. Moreover, two projects (i.e., Catroid and dex2jar) exhibited similar patterns where the instability of the releases analyzed decreased until they stabilized. In the case of SonarQube, the trend is to increase, maybe caused because SonarQube is one of the most active projects as developers continue adding new capabilities to the tool. The case of Hadoop is radically different as after a period where the instability of the releases is almost stable, starting to decrease, maybe caused by the removal of classes. Therefore, refactorings of projects help to stabilize or decrease the a bit instability of the project because they exhibit a better topology and the instability values can be kept under control.

Regarding the results of RQ3, in three of the projects analyzed (i.e., Sonarqube, dex3jar, and Hadoop), the instability of the package structure at a certain level is higher than the instability of the release. For instance, for dex2jar-0.0.9.15, the instability at level 5 shown in table 16 is higher than the instability of that release (i.e., 0.652549). This phenomenon is caused because the bigger number of dependencies is due to an increment in the number of packages in deeper levels of the package structure. In this way, we can advise software architects to not trust only in the instability values provided by the levels of the package structure, and at a certain level defined by the software architect, compute the instability of the overall project to check the possible deviations.

Finally, we can say that the architectural implications of instability variations are several. First, as the architecture evolves in terms of complexity, the instability tends to grow until a certain level where we observes some stabilization or even some decrease. Second, different topologies of the package and class structure may lead to different instability values, where we noticed that the instability of the architecture can decrease according to a different organization of their elements. Sometimes the instability values vary in small increments between two consecutive releases, as for large projects the increments or changes in the number of classes and relationships is not so big to produce bigger instability changes.

6. Threats to Validity

Here we describe the threats to the validity of our work.

Construct validity indicates the degree that measures what we claim in the experiments and on the relation between the theory and the observations [32] [33]. Based on well-established formulas, we believe our observations are correct based on the results computed for RQ1 and RQ2 and on the data mined from the repositories based on a common protocol. Only in the case of RQ2, we found a weak causal relationship to anticipate to instability values. At present, it is hard to mitigate this threat for RQ2, so new predictors of instability, maybe based on probability, are needed. For RQ1 we found correlations between the instability and the number of classes and dependencies (a.k.a. statistically conclusive validity) in half of the projects, so as future work we need to analyze more releases to find if there is any possible correlation between the instability of the project and the elements added and removed.

Internal validity indicates whether the experimental results soundly support the claims. Our findings prove partially a potential correlation between the instability and the changes in classes and edges, but this is not fully clear for all projects as in some cases the variation of the architectural instability between consecutive releases is small. For the future, we need to provide more accurate estimators able to predict the instability trend even if the number of releases is not high.

External validity indicates whether the findings generalize the results to different settings and populations. From our observations, we can generalize our results partially to other open-source projects as in the first research question we are just using accepted formulas, and the double check performed in the case of the Catroid confirms our results are correct. However, we acknowledge that the number of releases per project examined is not big and we need more observations to derive more solid findings, but sometimes it is hard to find more project releases or suitable to be processed by the tools used in this research. Moreover, computing additional tests for RQ3, adding third-party classes when we observed deviations of the instability values between the package levels and the project release can help to recalibrate our approach.

7. Related Work

Estimating the ripple effect of changes in OO classes has been analyzed by [34], where the authors suggested metrics such as the Number of Children (NOC) and Coupling between objects (CBO) to evaluate the scope of a change in classes. Closer to the architecture level, Diaz et al. [35] evaluate the impact of changes in product line architectures, while the authors in [36] classify 23 change impact analysis techniques for different software artifacts. Other authors like [37] described a ripple effect formula to predict the effect of changes in classes using coupling metrics. Few works investigated the use of instability metrics in software architecture. Ampatzoglou et al. [27] suggested an instability metric using probabilistic models to estimate the impact of changes in design patterns. The authors analyzed the stability of changes in classes using change proneness measures (i.e., a priori estimation) and instability measures (i.e., a posteriori measures). An extension of the aforementioned work is described in [38], where the authors proposed a method for assessing change proneness in classes due to evolving requirements, bug fixing, and ripple effect.

Regarding software instability, in [39] the authors investigate the instability based on afferent and efferent coupling metrics in open source projects, and they performed and statistical analysis of the results observing that 48% of software product had a high instability. In addition, Aversano et al. [24] analyzed the instability of architecture core components across releases and they defined instability metrics based on the packages that are added, removed, or changed. The authors in [40] analyzed the architectural stability of self-adaptive systems to achieve stable adaptations, and where instability can be considered an indicator of the sustainability of the system and architecture as well. Although works like [27, 38] highlighted that instability and change proneness measures a clear effect on the stability of a system as an indicator of its sustainability, only seminal works from [41, 42] suggested metrics to estimate the sustainability of architectural decisions.

Other recent works like [43] investigated the evolution of the instability in architectural smells across 524 versions in 14 open-source projects using the Arcan [9] tool. The approach uses a similarity index to measure the percentage

[9] `https://gitlab.com/essere.lab.public/arcan`

of elements that are shared by two sets affected by smell and hence, compute the smell density per component. The smells can be detected using the notion of instability gap as described in [44]. Also, in [45] the authors studied the historical class stability exploiting change history information as a way to predict unstable classes in 10 open-source projects and based on its correlation with change propagation factors. In [46] the authors provide an updated survey on the notion of stability in software engineering practice which is understood as long-term property for analyzing software evolution. The authors discussed the characterization and use of stability in software engineering research and the important quality attributes related to it, including sustainability. Also, from the different stability dimensions investigated, the authors highlighted the role of engineering practices for the evaluation of architectural stability, and they summarize the most popular stability metrics. In their findings, they claim that architectural decisions should be seen as planning for stability. Finally, the work from [19] discusses a similar approach to ours in terms of an analysis of the instability of open-source projects but the goals of the topics investigated are different from our work but somehow related to our research question 3.

8. Conclusions

This research reports a new insight based on the analysis of the instability values in four open-source projects. We provided instability values of different releases for the projects analyzed and we compared our results with some of the existing formulas, as we found some deviations of the values suggested in one of the approaches examined in the related work compared to ours. As we double-check the values computed for the elements added and removed and those shared between two consecutive releases, we ensure the results computed are the right ones and we provide additional insight about why the instability of a given release is not the same if we compute the instability of the previous release and the instabilities for the elements added and removed.

Moreover, the evolution of the instability discussed for RQ2 sheds light on the trend of how stable or unstable open-source projects are. Projects like SonarQube that have frequent changes or add new functionality exhibit and higher instability trends. On the opposite side, a reduction of the instability values often comes from major refactoring processes in the project. In addition, the proposed formula using the ratio of modified files contributes to the body of knowledge as we can use to estimate the instability changes in subreleases more accurately and warn developers about potential instability deviations.

From the results of our third research question, we provided a way to approach the final instability of a given release using the packages and classes reversed using a tool (i.e., ARCADE in our case) as we provided insights about the relationship between the complexity of the package structure and the instability values, also confirmed by the statistical tests. In addition, software architects and developers that would like to use our approach only need to count with a significant number of versions of their projects to provide more accurate estimators of the correlation between changes in classes and their relationships and the resultant project instability. As future work, we plan to evaluate how the instability affects other nonfunctional attributes, such as maintainability and evolvability, and provide an analysis for sample project releases in favor of the accuracy of our initial estimations. In addition, we plan to investigate the impact of the refactorings in the commit to suggest better ways to reduce the instability of ongoing projects.

References

[1] D. M. Le, C. Carrillo, R. Capilla, N. Medvidovic, Relating architectural decay and sustainability of software systems, in: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE, 2016.

[2] R. Capilla, E. Y. Nakagawa, U. Zdun, C. Carrillo, Toward architecture knowledge sustainability: Extending system longevity, IEEE Software 34 (2017) 108–111.

[3] N. H. Madhavji, J. C. Fernández-Ramil, D. E. Perry (Eds.), Software Evolution and Feedback, John Wiley & Sons, Ltd, 2006.

[4] S. Sehestedt, C.-H. Cheng, E. Bouwers, Towards quantitative metrics for architecture models, in: Proceedings of the First International Conference on Dependable and Secure Cloud Computing Architecture - DASCCA 2014, ACM Press, 2014.

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

[6] M. Lippert, S. Roock, Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, Wiley, 1 edition, 2006.

[7] N. Brown, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, Managing technical debt in software-reliant systems, in: Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER 2010, ACM Press, 2010.

[8] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, IEEE Software 29 (2012) 18–21.

[9] E. Bouwers, J. Visser, A. van Deursen, Criteria for the evaluation of implemented architectures, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009.

[10] H. Koziolek, Sustainability evaluation of software architectures, in: Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS 2011, ACM Press, 2011.

[11] P. Lago, S. A. Koçak, I. Crnkovic, B. Penzenstadler, Framing sustainability as a property of software quality, Communications of the ACM 58 (2015) 70–78.

[12] B. Penzenstadler, H. Femmer, A generic model for sustainability with process- and product-specific instances, in: Proceedings of the 2013 workshop on Green in/by software engineering - GIBSE 2013, ACM Press, 2013.

[13] C. C. Venters, L. Lau, M. K. Griffiths, V. Holmes, R. R. Ward, C. Jay, C. E. Dibsdale, J. Xu, The blind men and the elephant: Towards an empirical evaluation framework for software sustainability, Journal of Open Research Software 2 (2014).

[14] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, Measuring architecture sustainability, IEEE Software 30 (2013) 54–62.

[15] R. Martin, OO Design Quality Metrics - An Analysis of Dependencies, in: Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94.

[16] S. Black, Computing ripple effect for software maintenance, Journal of Software Maintenance 13 (2001) 263–279.

[17] T. L. Alves, J. Hage, P. Rademaker, A comparative study of code query technologies, in: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, pp. 145–154.

[18] M. Alenezi, F. Khellah, Architectural stability evolution in open-source systems, in: Proceedings of the The International Conference on Engineering & MIS 2015, ICEMIS '15, Association for Computing Machinery, New York, NY, USA, 2015.

[19] J. J. A. Baig, S. Mahmood, M. Alshayeb, M. Niazi, Package-level stability evaluation of object-oriented systems, Information and Software Technology 116 (2019) 106172.

[20] M. Alshayeb, M. Naji, M. O. Elish, J. Al-Ghamdi, Towards measuring object-oriented class stability, IET Software 5 (2011) 415–424.

[21] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, E. D. Nitto, Arcan: A tool for architectural smells detection, in: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 282–285.

[22] M. K. Chawla, I. Chhabra, Sqmma: Software quality model for maintainability analysis, in: Proceedings of the 8th Annual ACM India Conference, Association for Computing Machinery, New York, NY, USA, 2015, p. 9–17.

[23] D. Threm, L. Yu, S. Ramaswamy, S. D. Sudarsan, Using normalized compression distance to measure the evolutionary stability of software systems, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp. 112–120.

[24] L. Aversano, D. Guarda, M. Tortorella, Analyzing the instability of the core components of software projects, in: Proceedings of the 51st Hawaii International Conference on System Sciences, Hawaii International Conference on System Sciences, 2018.

[25] W. Li, L. Etzkorn, C. Davis, J. Talburt, An empirical study of object-oriented system evolution, Information and Software Technology 42 (2000) 373 – 381.

[26] D. Rapu, S. Ducasse, T. Girba, R. Marinescu, Using history information to improve design flaws detection, in: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., pp. 223–232.

[27] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, P. Avgeriou, The effect of GoF design patterns on stability: A case study, IEEE Transactions on Software Engineering 41 (2015) 781–802.

[28] M. Alshayeb, W. Li, An empirical study of system design instability metric and design evolution in an agile software process, Journal of Systems and Software 74 (2005) 269 – 274.

[29] C. Carrillo, R. Capilla, Ripple effect to evaluate the impact of changes in architectural design decisions, in: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA 2018, Madrid, Spain, September 24-28, 2018, pp. 41:1–41:8.

[30] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2009) 131–164.

[31] R. K. Yin, Case Study Research Design and Methods (5th ed.), Sage, 2014.

[32] R. Feldt, A. Magazinius, Validity threats in empirical software engineering research - an initial survey, in: Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010, Knowledge Systems Institute Graduate School, 2010, pp. 374–379.

[33] P. Ralph, E. D. Tempero, Construct validity in software engineering research and software metrics, in: A. Rainer, S. G. MacDonell, J. W. Keung (Eds.), Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering, EASE2018, Christchurch, New Zealand, June 28-29, 2018, ACM, 2018, pp. 13–23.

[34] N. Mansour, H. Salem, Ripple effect in object oriented programs, J. Comp. Methods in Sci. and Eng. 6 (2006) 23–32.

[35] J. Díaz, J. Pérez, J. Garbajosa, A. L. Wolf, Change impact analysis in product-line architectures, in: Software Architecture, Springer Berlin Heidelberg, 2011, pp. 114–129.

[36] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Software Testing, Verification and Reliability 23 (2012) 613–646.

[37] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, Introducing a ripple effect measure: A theoretical and empirical validation, in: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2015.

[38] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, A method for assessing class change proneness, in: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering - EASE 2017, ACM Press, 2017.

[39] D. Santos, A. M. P. de Resende, E. C. Lima, A. P. Freire, Software instability analysis based on afferent and efferent coupling measures, J. Softw. 12 (2017) 19–34.

[40] M. Salama, R. Bahsoon, Analysing and modelling runtime architectural stability for self-adaptive software, Journal of Systems and Software 133 (2017) 95–112.

[41] C. Carrillo, R. Capilla, O. Zimmermann, U. Zdun, Guidelines and metrics for configurable and sustainable architectural knowledge modelling, in: Proceedings of the 2015 European Conference on Software Architecture Workshops - ECSAW 2015, ACM Press, 2015.

[42] C. C. Venters, R. Capilla, S. Betz, B. Penzenstadler, T. Crick, S. Crouch, E. Y. Nakagawa, C. Becker, C. Carrillo, Software sustainability: Research and practice from a software architecture viewpoint, Journal of Systems and Software 138 (2018) 174–188.

[43] D. Sas, P. Avgeriou, F. Arcelli Fontana, Investigating instability architectural smells evolution: an exploratory case study, in: 35th Interna-

tional Conference on Software Maintenance and Evolution, IEEE, 2019.

[44] F. A. Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, pp. 433–437.

[45] S. Hussain, H. Afzal, M. Rafiq Mufti, M. Imran, A. Amjad, B. Ahmad, Mining version history to predict the class instability, PLoS ONE 14 (2019) 1–21.

[46] M. Salama, R. Bahsoon, P. Lago, Stability in software engineering: Survey of the state-of-the-art and research directions, IEEE Transactions on Software Engineering (2019).