Investigating instability architectural smells evolution: an exploratory case study

Darius Sas, Paris Avgeriou
Bernoulli Institute for Mathematics,
Computer Science and Artificial Intelligence
University of Groningen
Groningen, Netherlands
{d.d.sas, p.avgeriou}@rug.nl

Francesca Arcelli Fontana
Dipartimento di Informatica
Sistemistica e Comunicazione
University of Milano-Bicocca
Milan, Italy
francesca.arcelli@unimib.it

***Abstract*—Architectural smells may substantially increase maintenance effort and thus require extra attention for potential refactoring. While we currently understand this concept and have identified different types of such smells, we have not yet studied their evolution in depth. This is necessary to inform their prioritisation and refactoring. This study analyses the evolution of individual architectural smell *instances* over time, and the *characteristics* that define these instances. Three different types of architectural smells are taken into consideration and mined from a total of 524 versions across 14 different projects. The results show how different smell types differ in multiple aspects, such as their growth rate, the importance of the affected elements over time in the dependency network of the system, and the time each instance affects the system. They also cast valuable insights on what aspects are the most important to consider during prioritisation and refactoring activities.**

I. Introduction

In recent years, there has been increasing interest on the concept of *architectural smells* (AS): issues in the architecture that often cause extra maintenance effort [1]. Several studies have explored this concept and identified different types of such smells [1], [2], [3], [4]. However, while the evolution of *code* smell instances has been extensively investigated, very few studies focus on the evolution of *architectural* smells and do so only at a coarse-grained level (e.g. by simply counting the number of smells in each version). There is also no work tracking the individual smell instances along system evolution.

We need to study the evolution of AS in detail because AS are a different type of "affliction" than code smells: they usually involve more elements than code smells, they affect the system at a different scale, and they require more effort to be refactored [1]. At the same time, the long-term advantages of this refactoring in terms of maintainability and changeability of the system are higher. Thus, the current theoretical knowledge on code smells cannot be applied to AS.

In this study, we propose an approach to study the evolution of AS detected by an open source tool named Arcan [5], by tracking individual smell instances and measuring the evolution of the properties of each detected instance. We have detected almost 150.000 unique smell instances in over 500 versions across 14 open source Java projects. We have performed four types of analyses: a generic data mining analysis to have a better understanding of the data, a trend analysis to understand the evolution of the smells over time, a correlation analysis to identify possible correlations among the smell characteristics[1] considered, and a survival analysis to document their probability to persist within the system. The focus of this study is on the architectural smells known as instability AS [6]; these are introduced in more depth in Section III.

Our findings can enable practitioners and researchers to develop strategies for optimal refactoring prioritisation of individual smell instances based on multiple factors. For example, a Hublike dependency smell is a much better option for refactoring than a Cyclic dependency, especially in terms of complexity, and future and present maintenance effort. Additionally, Cyclic dependencies have a much shorter lifetime on the average, making them less critical in general.

The remainder of this paper is organised as follows: Section II discusses similar work in the literature, Section III introduces the smells hereby considered, Section IV explains the methodology of this case study, Sections V, VI, VII, and VIII report and discuss the results of the different analyses, Section IX lists the threats to the validity of this study and finally Section X concludes the paper.

II. Related work

We present related work concerning both architectural smells and code smells.

In the former case, Al-Mutawa et al. [7] have investigated the circular (or cyclic) dependencies' shape in Java programs. They developed and validated a methodology to detect and classify circular dependencies starting from the bytecode of an application. Their findings, based on a case study performed on the Qualitas Corpus [8] data set, suggest that the most common shapes (see Figure 2) are tiny and multi-hub. Moreover, they also argue that cycles among parents and children packages are less critical than cycles among non-related packages, providing empirical evidence to back up their claims.

[1] See Section III-B for the definition of characteristics and the full list.

Another study that considers the history of architectural smells was published by Roveda et al. [9]. In their work, the authors try to estimate the architectural debt index using architectural smells and track the evolution of the index throughout a system's history. The calculation uses partial historical information of the AS identified by the Arcan tool in multiple versions. The major shortcomings of Roveda et al.'s index are: (i) the historical information used is limited to the size of the smell and only considers the previous version, (ii) the historical information is weighted equally for every smell type, and (iii) it does not account for the *magnitude* of the variation, i.e. a decrease by only one element halves the contribution of the smell to the overall index, whereas an increase by only one doubles it. Indeed, one of the goals of this work is also to provide theoretical background and practical tools to improve such types of calculation.

Concerning code smells, there are several works on tracking smells throughout a system's history. In their work, Vaucher et al. [10] have focused on the code smell God Class and its evolution in terms of the degree of *"godliness"*, estimated using their previous approach based on Bayesian belief networks. The authors analysed the trend of such a parameter for each God Class instance in the history of two systems. Their findings suggest that the godliness of God classes tends to remain constant in over 60% of the cases.

A different perspective on code smells evolution was introduced by Chatzigeorgiou et al. [11], who analysed the survival probability of four types of code smells. Their findings show that Long Methods are the most persistent code smells in the two analysed systems.

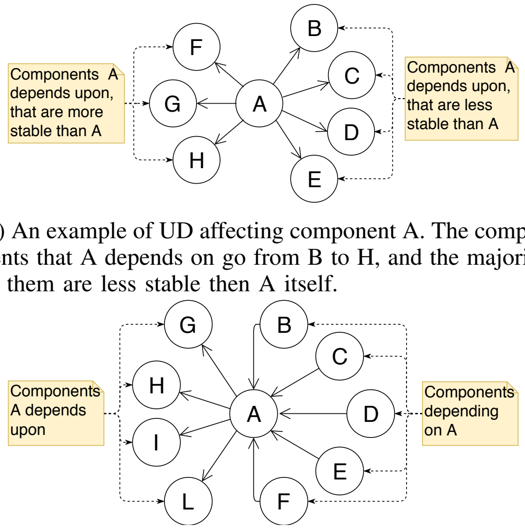
In a similar work, Peters et al. [12] have also analysed the persistence of code smells in a system, though they have used a slightly less elaborate technique to do so and on a slightly different set of smells. Their findings show that Feature Envy methods are the least persistent type of smell (similarly to the finds of Chatzigeorgiou et al.) and that Data Classes are, instead, the most persistent ones.

III. Architectural smells

A. *Definitions and implications*

This section lists the architectural smells (AS) considered by this study. The definition of these smells is provided by Arcelli et al. [6] and briefly reported here.

1) Unstable dependency (UD): This smell represents a component[2] that depends upon a significant number of components that are less stable than itself. The stability of a component is measured using Martin's instability metric [13], which measures the degree to which a component (e.g. a package) is susceptible to change based on the classes it depends upon and on the classes depending on it. The smell thus arises when a component has a significant number of components – the tool Arcan uses a 30% threshold [5] – it depends upon with an instability value higher than its own.



Components A depends upon, that are more stable than A
F
G
H
A
B
C
D
E
Components A depends upon, that are less stable than A

(a) An example of UD affecting component A. The components that A depends on go from B to H, and the majority of them are less stable then A itself.

Components A depends upon
G
H
I
L
A
B
C
D
E
F
Components depending on A

(b) An example of HL affecting component A, causing all of the components depending on A to be more susceptible to changes due to possible ripple effects propagating from the components that A depends upon.

Fig. 1: Examples of UD and HL smells.

A UD smell is detectable on Java package-like elements only (i.e. containers of classes). A simplified example of UD is shown in Figure 1a.

The main problem caused by UD is that the probability to change the main component grows higher as the number of unstable components it depends upon grows accordingly. This increases the likelihood that the components that depend upon it (not shown in Figure 1a for simplicity) change as well when it is changed (ripple effect), thus inflating future maintenance efforts.

2) Hublike dependency (HL): This smell represents a component where the number of ingoing and outgoing dependencies is higher than the median in the system and the absolute difference between these ingoing and outgoing dependencies is less than a quarter of the total number of dependencies of the component [6]. A hublike dependency can be detected both at the package and at the class level.

The implications of this smell for development activities are once again concerning the probability of change and the ease of maintenance. Consider, for example, the case represented in Figure 1b. Making a change to any of the components that A depends upon may be very hard [13], even though there is only one component depending on them. Additionally, the central component is also overloaded with responsibility and has a high coupling. This structure is thus not desirable, as it increases the potential effort necessary to make changes to all of the elements involved in the smell.

3) Cyclic dependency (CD): This smell represents a cycle among a number of components; there are several software design principles that suggest avoiding creating such cycles [1], [14], [15], [16]. Cycles may have different topological shapes. Al-Mutawa et al. [7] have identified 7 of them; the

[2] Generally, by *components* we refer to both classes and packages. Only in the case of UD, we only mean packages.

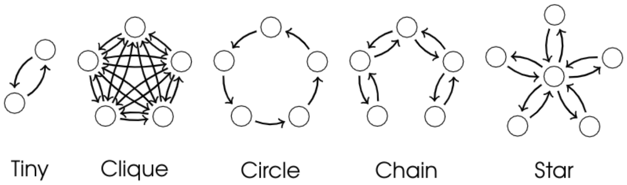


Fig. 2: Cycle dependency shapes. Figure originally published by Al-Mutawa et al. [7].

ones detected by Arcan are shown in Figure 2 [5]. Usually, the circle shape is intuitively perceived as the typical CD, but it is certainly not the only possible type of CD. In fact, there is empirical evidence [7] that tiny and multi-hub shapes (two stars attached together that are missing some edges) are more common than circle.

Besides affecting complexity, their presence also has an impact on compiling (causing the recompilation of big parts of the system), testing (forcing to execute unrelated parts of the system, increasing testing complexity), or deploying (forcing developers to re-deploy unchanged components) [1].

B. Architectural smell characteristics

An architectural smell characteristic is a property or attribute of an architectural smell instance. An architectural smell instance is a concrete occurrence of a type of architectural smell. For each architectural smell type, one can measure different characteristics. We refer to the characteristics that can be measured for every type of smell as *smell-generic*, whereas we refer to the characteristics that can only be measured for certain types of smells as *smell-specific* characteristics. The characteristics considered in our work are reported in Table I.

We decided to focus our analysis on this set of smell characteristics because they are measurable dimensions for the different facets of smells that further quantify the extent to which the smell affects the system; this can inform developers on how to prioritize refactoring. Additionally, some of the selected characteristics were developed, studied or discussed by other authors in previous studies, as reported by the Ref. column in Table I.

The smell-generic characteristic *Overlap*, *Centrality*, and *Size* are of interest because they are all metrics that are conceptually related to the complexity caused by any instance of a smell in the system. Intuitively, all of them *may* hinder the degree of understandability, extensibility, or generally of maintainability of the components affected by a smell: the more elements a smell has (size), or the more elements of a smell are also involved in other smells (overalp), or the more its elements are interacting with other important components of the system (centrality), the harder it is to fully understand or to refactor the smell.

Age, on the other hand, allows us to track the evolution of the other characteristics over time, identify periods where they are more impactful, or discern eventual correlations between them.

TABLE I: The smell characteristics identified by this study. * indicates this study. † marks characteristics not studied in this study as they are intended as future work.

Smell Character.		**Description**	**Ref.**
		smell-generic	
All	Age	The number of versions affected by the smell.	*
	Overlap Ratio	The ratio of the total number of components of a given smell that also take part in another smell.	*
	Centrality	The importance of the components affected by the smell within the system. Measured using the PageRank of the components in the dependency graph.	[9]
	Size	The number of elements of the system affected by the smell.	*
	Number of edges	The number of dependency edges among the components affected by the smell.	*
		smell-specific	
CD	Shape	The cycle shapes as shown in Figure 2.	[6], [7]
	Average edge weight	The number of dependencies (weight) between the components affected by the smell. It can be indicative of the difficulty of refactoring the cycle.	[5]
	Number of inheritance edges	The number of edges in the smell that represent an inheritance between components.	[17]
	Affected design level	Whether the cycle is present only at architectural level (among packages) or also at design level (among classes) too.	[7]
	Parent centrality†	The degree to which a package is at the centre of a cycle with its children sub-packages.	[7]
UD	Instability gap	The difference between the instability of the main component and the average instability of the dependencies less stable than the component itself.	[5]
	Strength (or DoUD [5])	The ratio between the number of dependencies that point to less stable components and the total number of dependencies of the class.	[5]
HL	Average internal path length†	Only computed on package HL. The average length of the paths between internal nodes with afferent dependencies and internal nodes with efferent dependencies within the central package. The shorter the length, the more the packages that depend upon the main component and packages that are depended upon by it are connected.	*
	Affected classes ratio†	Only computed on package HL. The ratio between the number of classes taking part in a dependency relationship with afferent and efferent packages of the main component and the total number of classes in the main component.	*, [18]

The CD smell-specific characteristics *Shape* and *Average edge weight* are of interest because they are directly related to the complexity of the smell. The more complex the shape, and the more edges there are between the affected components, the harder the smell is to refactor because more effort is required. The *Affected design level*, similarly, is important because the cycles present at both package and class level have an impact on two different levels at once. Finally, the *Number of inheritance edges* characteristic is considered because inheritance edges are considered an indicator of an intentional design choice [17], thus intentional cycles that contain a high number of inheritance edges between the components may be more interesting for a developer to inspect.

The UD smell-specific characteristic *Instability gap* and *Strength* are of interest because they are used for the detection of the smell and thus can effectively measure its criticality. The higher the instability gap, the higher the chance the component affected by the smell is changed due to ripple effects [13]. Likewise, the higher the strength, the higher the chance (because there are more possible components that are prone to a change) a change occurs and propagates to the affected component.

The HL smell-specific characteristics *Affected classes ratio* and *Average internal path length* are of interest because they quantify the involvement of the internal classes in the smell by answering the questions 'How many classes belonging to the affected package (out of all package classes) contribute to the smell?' and 'How much efferent and afferent packages are actually connected?', respectively. Intuitively, if the average internal path length is low, it is easier for changes to propagate through the components involved in the smell. And if the efferent and afferent packages are poorly connected (i.e. few paths), the chance a change propagates is small. In other words, these two characteristics measure the proneness of a HL smell to propagate changes incoming from its dependencies to the components depending upon it.

IV. Case study design

The design of the case study follows the guidelines proposed by Runeson et al. [19] to conduct and report case studies. Furthermore, the protocol used to conduct the study and keep track of the changes is based on the template proposed by Brereton et al. [20].

A. Goal and research questions

The objective of this study is to expand the current knowledge of architectural smells evolution. Using the Goal-Question-Metric [21] approach, the objective formulation is:

> ***Analyse*** *the evolution of individual architectural smells instances throughout the system's history* ***for the purpose*** *of understanding them* ***with respect to*** *their characteristics and lifespan* ***from the point of view of*** *software architects* ***in the context of*** *open source systems.*

Each one of the research questions that further refine the goal of this study focuses on a different aspect of their evolution: RQ1 studies the evolution trend of each type of smell w.r.t their characteristics, whereas RQ2 studies the survivability (or persistence) of each smell type. The two research questions (RQ1 and RQ2) are answered by answering a number of sub-questions (e.g. RQ1a and b for the case of RQ1).

RQ1 How does each type of architectural smell evolve throughout the system's history?

a) How do the smell characteristics of each smell type (i.e. size, centrality, etc.) evolve over time?
b) Is there a correlation between smell characteristics of the same smell type?

This research question focuses on investigating the evolution of each type of architectural smell through their characteristics and identifying relations among them. It can provide information for understanding the effects of each type of smell on the system, which can then be used to define refactoring prioritisation rules based on single instances of that type. Identifying relations is important to avoid using the same information multiple times. This means that it is necessary to identify eventual correlations among them so that we can determine if we can omit some of the characteristics without losing essential information.

One example of the use of trend as indicator for extra maintenance effort could be the trend of *centrality*, a smell-generic characteristic that measures the degree of connectivity of the elements affected by a smell with the other system's components: the higher the values the more other components are in some way connected to it and thus the more probable for a change to have ripple effects.

RQ2 How do the different types of smells compare against each other regarding their lifespan?

a) Which types of smells, CD, HL or UD, are more persistent (i.e. are less common to be removed)?
b) Do the same smell types at package and class level have a different survival probability?
c) Does the shape of a CD smell affect its lifetime?

The aim of this research question is to compare the different types of smells in terms of their survivability. Answering this question could help to define prioritisation rules at the level of smell type. For example, one could choose to first refactor the types of smells that are more likely to persist longer within the system.

We decided to focus on survivability because it is a time-based measurable dimension of architectural smells, affecting future maintenance: the longer an AS affects a system, the longer the developers and architects will spend extra maintenance effort on the affected components.

B. Case selection

In this study, we used a set of open source systems known as the Qualitas Corpus (QC) [8]. We decided to work with open source systems (OSS) for the following reasons: OSS are easy to retrieve and manipulate, the QC has a big variety of different projects ready to be used, and it is easier to develop static analysis tools when there is the possibility to inspect the source code analysed. We consider the extension of our analysis on industrial systems as future work.

The QC has more than 100 projects that can be potentially analysed. We required that projects have more than 15 versions available so to ensure smells have enough time to grow, evolve, and fade, thus limiting the number of candidate projects to 15. We also removed EclipseSDK from our selection due to its size causing difficulties during tracking. The demographics of the selected projects are shown in Table II.

TABLE II: The projects from the Qualitas Corpus release 20130901e used in this study. A total of 524 versions (both major and minor) were analysed.

Project	# Versions	First version	Last version	# Unique AS
Ant	23	1.1	1.8.4	1211
Antlr	22	2.4.0	4	1183
ArgoUML	16	0.16.1	0.34	3886
Azureus	63	2.0.8.2	4.8.1.2	108796
Freecol	32	0.3.0	0.10.3	13259
Freemind	16	0.0.2	0.9.0	994
Hibernate	115	0.8.1	4.2.2	13551
JGraph	38	5.4.4	5.11.0.1	249
JMeter	24	1.8.1	2.9	1846
JStock	30	1.0.6	1.0.7c	927
Jung	23	1.0.0	2.0.1	238
JUnit	24	2	4.11	164
Lucene	35	1.3.0	4.3.0	1126
Weka	63	3.0.1	3.7.9	2164

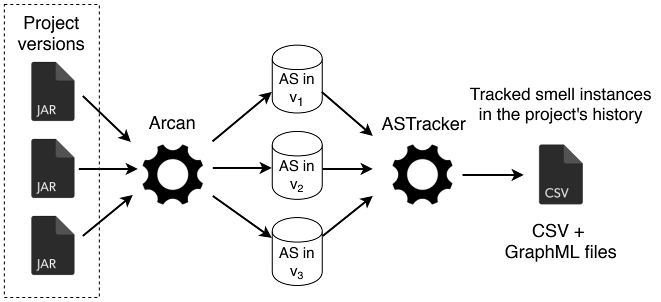


Fig. 3: Data collection process and tooling. The data of each individual project was then merged in a single data set.

C. Tooling

To perform the study, we developed a toolchain that allows to mine architectural smells from a series of precompiled Java systems, as illustrated in Figure 3. The toolchain is composed of two parts: AS *detection* and AS *tracking*.

1) Architectural smell detection: To identify architectural smells we use Arcan[3], a free Java tool for detecting architectural smells in a system. Arcan receives as input one, or multiple, JAR files of a single version of a system and outputs a series of CSV files and a GraphML file. The graph file is the dependency graph of the given system extended with nodes denoting architectural smells. The same information as the graph file is contained within multiple CSV files.

2) Architectural smell tracking: In order to perform the study, we needed to track the architectural smells for each pair of consecutive versions of the system, i.e. from v_1 to v_2, from v_2 to v_3, and so on. To this end, we developed a tool, ASTracker[4], that performs the following steps: it takes as input multiple versions of a system (the GraphML files produced by Arcan) and maps every smell in each version to its closest successor in the next version, calculates the smell characteristics, and returns the results as CSV and GraphML files.

To perform the mapping of each smell to its successor we use a function J known as *Jaccard similarity index* [22], defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

where A and B are the sets of the affected components in two consecutive versions. The index simply measures the percentage of elements that are shared by the two sets. The use of this methodology and of the Jaccard index are justified because a smell is defined by the elements it affects: the similarity of the affected sets of elements leads to identifying the successor of a smell.

The comparison among elements in the sets is made using the full name of the classes/packages. The main advantage of this method is that it avoids name conflicts; however, a renaming in any of the parent packages results in the inability to track the smell in the next version. Thus, for every smell k in version v_1 and for every smell l in version v_2, we compute $j_{kl} = J(a(k), a(l))$ which is basically a matrix where the rows are the smells from v_1 and the columns are the smells from v_2. The function a returns the set of elements affected by a smell. The linking between smells k and l is done using a **greedy strategy**: the highest j_{kl} such that k and l have not already been linked with another smell, is the next mapping $k \rightarrow l$ to be created. The greedy strategy ensures that every smell has been linked with the smell that is most similar to itself, which means formally that only one cell per row and column from the matrix j is selected. This operation is repeated until there are no more smells left to map or the similarity scores of the remaining ones do not satisfy $j_{kl} \geq \theta$, where θ is the similarity threshold defined as

$$\theta = \begin{cases} 0.67 & \text{for } |a(k)| > 5 \\ 0.60 & \text{for } |a(k)| \leq 5 \end{cases}$$

We selected a variable threshold in order to cover the big variance of the function J when $a(k)$ has a relatively small cardinality. To adjust the thresholds, we consulted all the possible values of J in the case where the two inputs shared all of their elements but only the size changed. Additionally, we also consulted all the possible values for small input sets sharing a variable number of elements. The selection of $\theta = 0.60$ when $|a(k)| \leq 5$ allows for a maximum difference of 3 elements with a smell's successor, allowing the algorithm to be more permissive for smells with fewer elements. Likewise, a value of $\theta = 0.67$ allows for a reasonable variation when the size of an AS is bigger than 5.

The algorithm only maps smells of the same type, namely CD with CD, UD with UD, and HL with HL.

V. General results

This section introduces some general statistics and insights concerning the data we have collected[5].

[3] See https://gitlab.com/essere.lab.public/arcan.

[4] See https://github.com/darius-sas/astracker to access the tool and the data used in this study.

[5] Supplemental material available at http://www.cs.rug.nl/search/uploads/Resources/supp-material-as-evo-icsme19.zip.

A. *Smell density*

A good starting point in understanding the evolution of smells is to look at the smell density (# of smells per component). As the smell density in a system gets closer to one it means that, on the average, there is one smell for every component in the system. Figure 4 shows the density of each smell type across the versions of every system.

Remarkably, seven projects have a smell density for CD among packages that is either higher or very close to 1 in most of their versions, meaning that it is quite common for developers to create cycles among the packages of the systems, thus increasing the complexity of the system. It is interesting though to note that the density of CD among classes, in most of the systems, is more or less constant throughout time despite the size of the systems growing (i.e. their ratio remains mostly constant). In other words, CD smells at class level are constantly introduced by developers as a by-product of the development activities as the system evolves. This causes also the number of cycles among packages to increase (because some of those cycles will be among classes from different packages), and since the number of classes per package increases over time in most of the systems analysed the smell density on packages is bound to increase as well. A similar pattern also emerges for UD smells, which are also constantly introduced in the system and have a growing trend. On the contrary, the number of HL smells stays mostly constant and relatively low (less than 10) over time in all the systems analysed, which is expected as a system has only few components that have a disproportionate number of dependencies.

Takeaway
Dependencies across packages affected by CD smells become ever tighter as the system ages, making it more difficult over time to reuse them seperately, without importing the whole system. This is caused because the cycles among packages grow in number at a higher rate than the number of packages itself.

B. *Smell characteristics*

In this section, we briefly cover some interesting findings on the characteristics mentioned in Table I. One noteworthy finding is the difference in size between smells. HL smells, due to their definition, tend to be usually bigger than the other types of smells, surpassing 100 elements in bigger systems, whereas UD smells are the smallest ones, hardly surpassing 10 elements even in bigger systems. However, CD and UD smells have higher overlap ratio in general, meaning that trying to refactor a smell with high overlap will entail also dealing with a certain number of other smells.

Concerning UD smells specifically, we note that their instability gap mostly ranges between -0.1 and -0.3; since these values are relatively close to zero, we argue that they are not very prominent and by slightly improving the instability of few packages, the smell could be removed. However, the instability gap is also decreasing over time for 50% of the UD smells detected (more details on this analysis in Section VI), meaning they become more severe over time.

Finally, we also note that CD smells are mostly at the class level only[6] (ranging from 60% to 95%, depending on the project) or package level only (from 0 % to 75%, depending on the project). A small percentage (less than 3%) affects class and package level at the same time and an even smaller percentage (1-2%) switch between levels over time (e.g. they go from class level only to both architectural and class).

VI. Trend analysis (RQ1a)

A. *Methodology: dynamic time warping*

Analysing the trend of all the characteristics of each smell instance detected in the analysed systems was not a trivial problem to solve, due to its dimensionality (smell instances, time, characteristic). The approach adopted to solve the aforementioned problem was signal classification: the values assumed by a certain characteristic for a certain smell over time are considered as a signal, then they are compared to a series of predefined signals and a label is assigned to each one of them based on the *distance* from each template. We used dynamic time warping[7] [23] to warp the signal of each template and stretch it to match the signal one desires to compare it with. This technique was previously used by Vaucher et al. to classify the trend of God Classes [10].

Formally, we can model the problem as follows: for every smell characteristic C^k of a certain smell k we consider the different values C^k_i as a signal S. We then compute the following variables: $h = \max S$; $l = \min S$; and $m = (h + l)/2$. These three values are then used to build the seven templates, named from a to g, as shown in Figure 5. For example, temblate (b) is defined as $b = (l, m, h)$. The templates are re-adjusted for each signal classified. Finally, the signal is classified by comparing the distance of the signal from each template, and selecting as a label the template name of the closest signal template. Specific implementation details can be inspected in the source code[4].

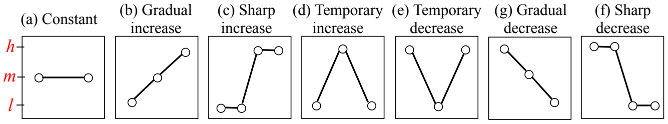


Fig. 5: Trend evolution classification templates. Figure adapted from the work of Vaucher et al. [10].

Despite the selected templates offering a good variety of possible signal shapes, there may be some cases that are not described well enough by the current selection. For example, signals that vary between two integer values (e.g. 6-7) multiple times, would be classified by the model as a constant signal (i.e. template (a)). Nonetheless, we deem that the approximation offered by the model when unusual signal curves have to be classified is sufficient for the purpose of this paper for the following reasons:

[6] See 'Affected design level' in Table I for more details.

[7] The implementation used for this analysis was provided by the R package `dtw`.

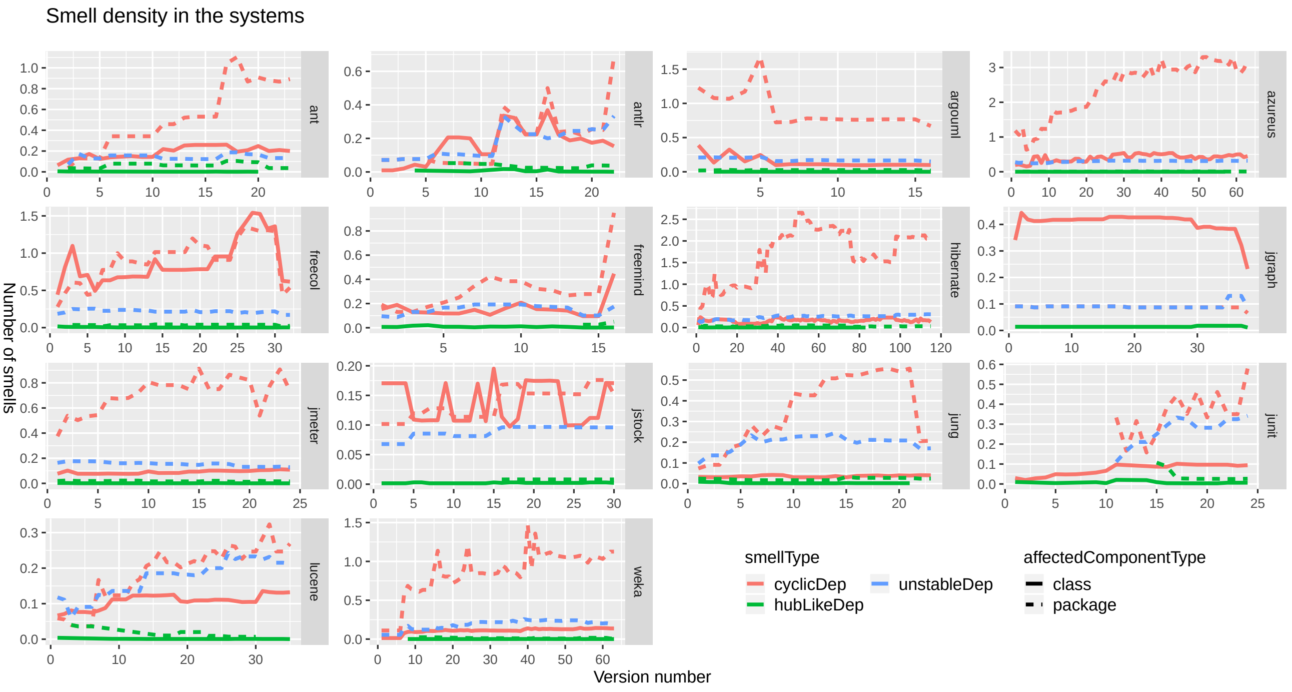


Fig. 4: Number of smells in the system divided by total number of classes or packages, depending on the type of the component affected.

- the templates selected represent simple and general cases, thus they simplify interpretation and analysis;
- a signal is classified based on the distance between points from the template and points from the signal itself after being warped, thus the classified signal has at least an internal component that resembles the classification tag (i.e. template) assigned.

B. Results

We performed the aforementioned analysis for all of the numeric characteristics we have recorded. Hereby we report only the most interesting ones, as there is a large number of data and results that could not realistically fit into this paper.

a) Size: Overall, the size of the smells stays constant throughout their evolution, especially in the case of CD and UD. This is shown in Figure 6 where approx. 50% of the total CD and UD across all systems have a constant trend. Instead of growing in size, CD smells tend to grow in number, spreading across the system as new elements are added to the system's dependency network (i.e. new classes, packages, etc.). Nevertheless, there is a fair amount of smells among all the types that exhibit an increasing trend of some kind (types B, C, D). Specifically, HL smells tend to grow in nearly 65% (40% Sharp and 25% Gradual increase) of the cases. Given its nature, having a hub that keeps getting bigger and bigger through dependencies from more and more classes, or packages, is problematic: that part of the system becomes more complex, it has a lower cohesion and a higher coupling, thus hindering future maintenance activities on it. It is thus important to *limit the growth* of such smells by redistributing the responsibility of the central component affected by the smell to others.

b) Number of Edges: Contrary to size, the number of edges connecting the components affected by a smell have a different trend: they tend to increase. Specifically, as can be seen in Figure 6, each smell type exhibits an increasing trend in the number of edges involved in the smell of at least 40% and up to 80%. Additionally, the number of edges between the affected components grows faster than the number of components per se. Again, this is especially true in the case of HL smells, making them the type of smell that grows faster among the smells studied in this work. Hublike dependencies are thus an important source of extra maintenance effort, and the number of edges among the affected components of an HL smell can quantify this effort more precisely than the number of affected elements. Indeed, this makes sense because an increasing number of edges between components also increases the probability that a change propagates to adjacent components that depend on the component subject to change (as described in Section III-A2). This fact was also mentioned in a previous work on change proneness metrics of software packages, where the number of method calls (and thus also dependencies) has been used as a change proneness indicator [24]. Additionally, Martin also links dependencies with change proneness [13].

c) Centrality: The centrality metric selected is PageRank [9]. We decided to measure the PageRank of a smell as the maximum PageRank value of the affected components and then weight it against the number of elements in each version. This weighting makes sense because as the system

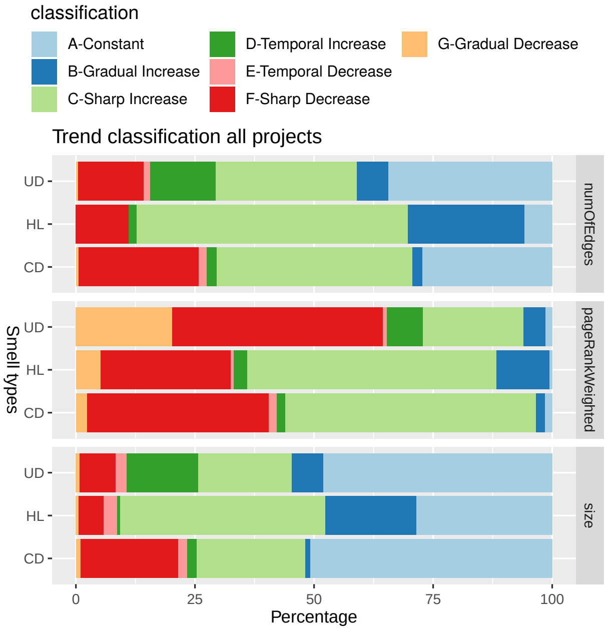


Fig. 6: Signal classification distribution for different characteristics aggregated for all projects as percentages of the total number of smells of that type.

ages, also the number of nodes in the graph used for the calculation of the PageRank increases, scaling down its values, but maintaining the proportions, hence the weighted version allows us to account for this phenomenon.

As one can observe in Figure 6, as the system and the smells age, the centrality of the smells tends to increase in the vast majority of the cases, especially for CD and HL. On the other hand, UD exhibit more or less the opposite trends.

The results indicate that the component with the highest PageRank (which is very likely to be the central component) in HL smell tends to *"move"* to the centre of the system as the system ages. A similar trend can be observed for CD smells too. These results confirm a very important assumption for these two types of smells: *AS tend to move to more central parts of the system as they age*. These central parts are also the most important as they have many ingoing dependencies. Consequently, increasingly more maintenance is required for the parts of a system that are affected by CD and HL smells.

Unexpectedly, for UD one can observe the opposite since most of them exhibit a decreasing trend (types E, F, and G).

Takeaway
Hublike dependency smells are a better target for refactoring activities in terms of reduction in complexity, future maintenance efforts, and ease of removal for refactoring activities are likely to focus mostly on the central component, by moving functionality elsewhere, rather than on several components as in the case of multiple CD smells.

VII. Correlation analysis (RQ1b)

To identify related *pairs of characteristics* for each smell instance of the same type, and for each pair of characteristics, we ran a Spearman correlation test to check for eventual correlations. The test was selected because the data is not normally distributed and it is not possible to assume that there is any linear relationship among all the characteristics neither. The test was performed on each smell instance and only on the pairs of smell characteristics whose both standard deviations were not equal to zero for that instance. The aggregate test results for all smells were plotted using boxplots (only $p \leq .05$). The plots are included in the supplemental material[5] for space reasons.

The characteristics that present a correlation for the majority of the instances detected are the following:

Num. of edges ∼ Overlap[8] for smells of type HL and CD at package level. This is expected because of the high smell density at package level (as shown in RQ1a). UD, however, do not present such a correlation for these characteristics; this is probably because they usually do not affect central parts of the system[5], which are more likely to be affected by multiple smells.

Num. of edges ∼ Centrality for HL smells at class level. This was also expected due to the definition of HL (i.e. a component with a lot of incoming and outgoing dependencies, which increases PageRank by definition). CD at class level also exhibit a correlation for these two characteristics, but a bit weaker, probably because CD are more frequent among elements near the center.

Num. of Edges ∼ Size strongly for all smells, which is expected.

Overlap ∼ Centrality only weakly. The most prominent correlation is for HL at class level, but is once again expected.

Overlap ∼ Size for CD at class and package level, is also expected, as the bigger the size, the more likely it is that the elements affected are also affected by other smells. The correlations also exist for HL smells, though they are a bit weaker.

Number of edges seems to be correlated with a number of characteristics in multiple cases. Despite this result, it is hard to state that, based on this correlation, one should ignore the other characteristics, as these correlations mostly refer to the *majority of the instances* rather than being an absolute gauge of the general case. In fact, the only pair of characteristics that one can state that are fully correlated for all smell types, independently of the instance, are Size and Number of Edges. The other correlations are either not valid for all of the smell types, or only a part of the instances analysed show solid evidence of correlation.

VIII. Survival analysis (RQ2a,b,c)

A. Methodology: the Kaplan-Meier estimator

The rate of survivability of an architectural smell within a system may drastically vary depending on its type. To establish the rates and compare them among the different projects and smell types, we employed a technique typically used in the biomedical sciences, in product reliability assessment, and also employed to analyse code smell persistence in previous studies [11]. Unlike simple descriptive statistics, such as mean, density functions, and similar, survival analysis also takes into consideration the possibility that a smell continues to affect the system even after the last version included in the analysis. In the biomedical domain, this event is associated with the patient surviving past the period of the analysis.

The survival analysis is accomplished using the Kaplan-Meier estimator [25], a non-parametric statistic that estimates the survival probability of a type of smell as the system evolves (new versions are released). The statistic gives the probability that an individual patient (i.e. smell in our case), will survive past a particular time t. At $t = 0$, the Kaplan-Meier estimator is equal to 1, and as t goes to infinity, the estimator goes to 0. Also, the probability of surviving past a certain point t is equal to the product of the observed survival rates until t.

B. Results

Figure 7 reports the results of the analysis, i.e. the survival probabilities (a) of different smell types and (b) of different cycle shapes.

1) Survival probabilities of different smell types (RQ2a,b): One pattern that emerges from Figure 7a is that CD smells fade much quicker than the other types of smells in almost all of the systems and have a very small probability to persist within the system for a long time. We conjecture that the cycles that persist the most are the cycles among the fundamental components of the system; these are very unlikely to change after the core development activities for that part settle down and new functionalities attract the effort of developers. Moreover, we also note that cycles only have a *50% chance* to stay within the system for more than 4-5 releases. Furthermore, cycles among classes persist a little longer within the system than cycles among packages, probably because classes taking part in cycles at design level only might have a stronger coupling with each other than packages.

Another pattern that emerges is that UD is the most persistent type of smell, being the one with the highest survival probability in the long run. Its survival probability is so high that in some systems it never falls below 50%, even when there are a lot of versions such as in the case of Azureus. Moreover, it also decreases at a much slower rate than the other types of smells, making it an ideal target for refactoring to avoid extra maintenance effort in the long run.

HL smells, are more or less in between the other two smell types. They exhibit a similar decrease rate in survival probability as CD smells but eventually end up surviving for more releases. However, this pattern does not hold for all the projects, and in some cases, HL smells end up being removed within fewer versions than CD. This trend holds true especially for HL at the class level, which tend to decay much faster than HL at the package level. Thus, it is reasonable to state that HL at package level can be prioritised over HL at class level as they have a higher chance of requiring extra maintenance over time. In general, from this analysis one can conclude that package level smells, such as UD and HL on packages, tend to last a little bit longer than class level smells, implying that smells at the package level are potentially more impactful on maintenance efforts than smells affecting classes only.

Takeaway
The refactoring prioritisation should **not focus on cyclic dependencies** that were recently introduced, as it is very likely that they will disappear within the next few releases because they are less likely to influence the maintenance effort on the long term. Instead, refactoring should first focus on either UD smells or HL smells among packages as they exhibit higher persistence rates. This also confirms that most circular dependencies are not critical [7].

2) Survival probability of different CD shapes (RQ2c): Concerning the different shapes of CD smells, Figure 7b shows how different shapes persist within the system. The results show that the most pervasive shape in most systems are tiny shapes. This makes sense as tiny shapes are composed by only two elements and there might be multiple dependency edges between the two elements; thus the probability of a tiny cycle to break is smaller than shapes with multiple elements. Additionally, tiny cycles are easier to understand and may also be intentionally designed as such.

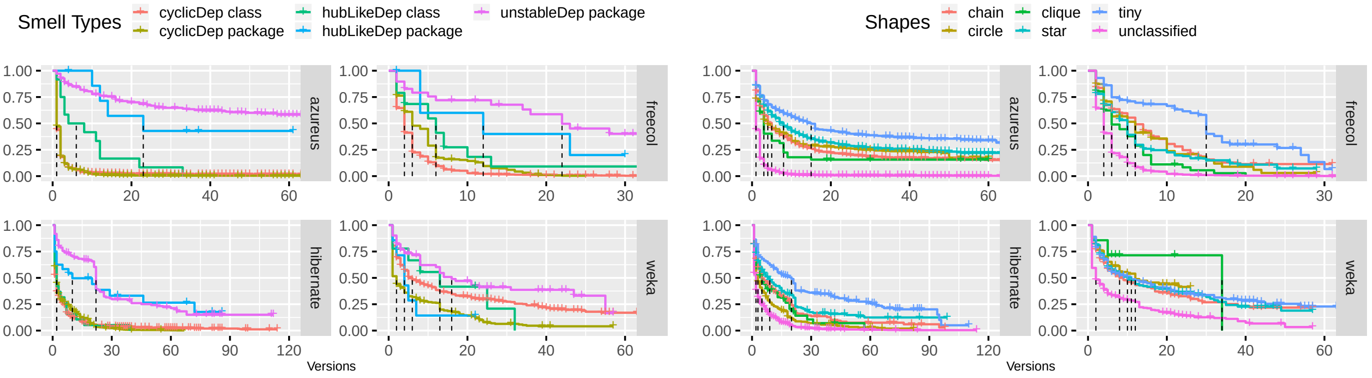
On the other hand, the other, more complex, shapes are less *resilient* (i.e. they disappear faster than tincy cycles) and there is very little difference between different shape types, making it hard to formulate any solid proposition on their survivability. In order for these complex shapes to persist, they must affect parts of the system that have a solid conceptual connection; otherwise they do not persist long within the system.

Regarding instead the cycles Arcan could not classify into definite shapes, they have a more consistent trend and disappear quicker than all other shapes. A possible explanation could be related to their nature: we conjecture that this type of cycle is mostly random and caused by casual relationships among components that tend to connect multiple uncomplete cycles into a single one, possibly overlapping with other cycles as well. Thus these very volatile edges that interconnect multiple parts of a system have a high chance of getting changed because they are individual edges, and if one of these edges is removed, the whole cycle breaks. This is also evident from the clear difference in survival probability between the unclassified shapes and the complex shapes (circle, chain, clique, star).

Takeaway
We suggest that tiny shapes should not to be prioritised during refactoring even though it is the most persistent one, as it may be the **result of intentional design** (false positives). Refactoring activities, instead, should prioritise old cycles with complex shapes that are more likely to affect important parts of the system, and thus that are more likely to incur extra maintenance effort.

IX. Threats to validity

We identified the possible threats to validity for this study and categorised them using the classification proposed by Runseson et al. [19]: *construct validity*, *external validity*, and *reliability*. Internal validity was not considered as we did not examine causal relations [19].



(a) Smells of different types affecting classes or packages.

(b) Cyclic dependencies of different shapes.

Fig. 7: Survival probability p up until any time t. $p = 0.50$ is represented by a vertical dashed line. Only a selection of systems is shown here for the sake of readability. The full plot is available in the supplemental material[5].

a) Construct validity: This aspect of validity reflects to what extent this study measures what it is claiming to be measuring [19]. In this study, we aim at measuring the evolution of architectural smells instances and understand them depending on their type and different characteristics. We developed a case study using a well-known protocol template [20] that was reviewed by the three authors and an external researcher in several iterations to ensure that the data to be collected would indeed be relevant to the research questions.

A possible threat to construct validity is the correctness of the tracking algorithm that might be incorrect or not cover some special cases, such as the renaming of the affected components. To mitigate this threat, we manually validated the tracking results for one of the projects considered in this study (Antlr) and fixed any issues we found during our inspections.

Another threat concerns the detection of the smells considered in this project which depends on the implementation offered by Arcan. This is partially mitigated, as the Arcan tool has already been used and evaluated in a number of studies [6], [26].

Finally, the last threat we identified is the relatively *long*, and *variable* periods of time in between the versions analysed for each project. This problem may have caused the prevalence of ‘sharp’ classification in the trend analysis over the ‘gradual’ ones. We mitigated this threat by limiting the importance we attribute to the specific type of the trend and focusing mostly on its nature (i.e. increase/decrease) and by also including projects with a strict release schedule (e.g. Hibernate).

b) External validity: This aspect of validity reflects to what extent the results obtained by this study are generalisable to similar contexts. The second one regards the projects we used to collect the necessary data. These projects were all open-source Java systems, Hence, it is not possible to generalise these results to industrial projects or projects written in a different programming language. However, we addressed this threat by adopting a collection of systems (the Qualitas Corpus) specifically intended for scientific analyses and tried to include as many *projects* and *versions* as possible in order to increase the sampling size of the population analysed. Our findings can thus be generalised to other Java projects of similar size and history that have an active open source community backing the development efforts.

c) Reliability: Reliability is the aspect of validity focusing on the degree to which the data and the analysis are dependent on the researcher performing them.

The data and the tools used in this study are freely available online[3,4] to allow other researchers to assess the rigour of the study or replicate the results using the same data set or even on a different set of projects.

The reliability of the findings is guaranteed by the fact that all the intermediary results were inspected by a second researcher during all the data analysis process. The analysis was also performed using well-established techniques already used in previous work for analysing similar artefacts (code smells) as well as also in different fields (e.g. survival analysis, in the biomedical sciences field).

X. Conclusions

This study has investigated the evolution of instability architectural smells in the context of open source systems with respect to their characteristics and persistence. We presented multiple findings and practical implications useful both for practitioners and researchers that can help them improving the strategies for reducing long term maintenance efforts by managing architectural smells.

As future work, we plan to extend our tooling to mine architectural smells directly from Git repositories, thus allowing us to link the current information to code churn and investigate the effects of smells on change rates.

Acknowledgements

This work was supported by the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 780572 SDK4ED (https://sdk4ed.eu/).

References

[1] S. R. Martin Lippert, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. 2006.

[2] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Identifying Architectural Bad Smells,” in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 255–258, 2009.

[3] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. 2014.

[4] R. Mo, Y. Cai, R. Kazman, and L. Xiao, “Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells,” *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*, pp. 51–60, 2015.

[5] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto, “Arcan: A tool for architectural smells detection,” *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pp. 282–285, 2017.

[6] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, “Automatic detection of instability architectural smells,” *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, pp. 433–437, 2017.

[7] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, “On the shape of circular dependencies in java programs,” in *Proceedings of the Australian Software Engineering Conference, ASWEC*, pp. 48–57, IEEE, apr 2014.

[8] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, “Qualitas corpus: A curated collection of java code for empirical studies,” in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pp. 336–345, Dec. 2010.

[9] R. Roveda, F. A. Fontana, I. Pigazzini, and M. Zanoni, “Towards an architectural debt index,” in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 408–416, IEEE, 2018.

[10] S. Vaucher, F. Khomh, N. Moha, and Y. G. Guéhéneuc, “Tracking design smells: Lessons from a study of God classes,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 145–154, IEEE, 2009.

[11] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of code smells in object-oriented systems,” *Innovations in Systems and Software Engineering*, vol. 10, pp. 3–18, mar 2014.

[12] R. Peters and A. Zaidman, “Evaluating the Lifespan of Code Smells using Software Repository Mining,” in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 411–416, IEEE, mar 2012.

[13] R. Martin, “Oo design quality metrics,” *An analysis of dependencies*, vol. 12, pp. 151–170, 1994.

[14] D. L. Parnas, “Designing software for ease of extension and contraction,” *IEEE transactions on software engineering*, no. 2, pp. 128–138, 1979.

[15] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured design,” *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[16] R. C. Martin, “Design principles and design patterns,” *Object Mentor*, 2000.

[17] J. Laval, J.-R. Falleri, P. Vismara, and S. Ducasse, “Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems.,” *The Journal of Object Technology*, vol. 11, p. 4:1, apr 2012.

[18] H. Abdeen, S. Ducasse, and H. Sahraoui, “Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software,” in *2011 18th Working Conference on Reverse Engineering*, pp. 394–398, IEEE, oct 2011.

[19] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering - Guidelines and examples*. 2012.

[20] P. Brereton, B. Kitchenham, D. Budgen, and Z. Li, “Using a protocol template for case study planning,” in *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*, no. 2006, p. 8, 2008.

[21] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, “Goal Question Metric (GQM) Approach,” in *Encyclopedia of Software Engineering*, 2002.

[22] P. Jaccard, “The distribution of the flora in the alpine zone,” *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[23] J. Kruskal and M. Liberman, “The symmetric time-warping problem: From continuous to discrete,” *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, 01 1983.

[24] E. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and 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)*, pp. 1–10, Oct 2015.

[25] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” *Journal of the American statistical association*, vol. 53, no. 282, pp. 457–481, 1958.

[26] A. Biaggi, F. Arcelli Fontana, and R. Roveda, “An Architectural Smells Detection Tool for C and C++ Projects,” in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 417–420, IEEE, aug 2018.