SN Computer Science (2021) 2:13
https://doi.org/10.1007/s42979-020-00407-5

ORIGINAL RESEARCH

Exploring the Relation Between Co-changes and Architectural Smells

Darius Sas[1] · Paris Avgeriou[1] · Ronald Kruizinga[1] · Ruben Scheedler[1]

Received: 1 July 2020 / Accepted: 13 November 2020

Abstract

The interplay between Maintainability and Reliability can be particularly complex and different kinds of trade-offs may arise when developers try to optimise for either one of these two qualities. To further understand how Maintainability and Reliability influence each other, we perform an empirical study using architectural smells and source code file co-changes as proxies for these two qualities, respectively. The study is designed using an exploratory multiple-case case study following well-know guidelines and using fourteen open source Java projects. Three different research questions are identified and investigated through statistical analysis. Co-changes are detected by using both a state-of-the-art algorithm and a novel approach. The three architectural smells selected are among the most important from the literature and are detected using open source tools. The results show that 50% of co-changes eventually end up taking part in an architectural smell. Moreover, statistical tests indicate that in 50% of the projects, files and packages taking part in smells are more likely to co-change than non-smelly files. Finally, co-changes were also found to appear before smells 90% of the times a smell and a co-change appear in the same file pair. Our findings show that Reliability is indirectly affected by low levels of Maintainability even at the architectural level. This is because low-quality components require more frequent changes by the developers, increasing chances to eventually introduce faults.

Keywords Architectural smells · Co-changes · Logical coupling · Empirical study

Introduction

The interplay between design-time (e.g. Maintainability) and runtime qualities (e.g. Reliability) can be particularly complex: when developers try to optimise for either one of these two types of qualities, important trade-offs may arise. For example, striving for a simple and maintainable design, might inevitably affect the system run-time performance or security. Likewise, aiming to build a speedy and dependable system might require to increase the system's inherent complexity, thus sacrificing its evolvability.

While there is currently a considerable amount of research effort to understand this interplay [5], the topic is still understudied and there are many aspects that have not been investigated. This paper focuses on one such aspect: the interplay between Reliability and Maintainability, specifically by studying co-changes and architectural smells as their proxies, respectively. We elaborate on both proxies in the following paragraphs.

When analysing the history of a system, co-changing files provide crucial insights on the implicit dependencies among files. Historically, co-changes are considered a sign

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

This article is part of the topical collection "Interaction between Energy Consumption, Quality of Service, Reliability and Security, Maintainability of Computer Systems and Network" guest edited by Erol Gelenbe.

✉ Darius Sas
d.d.sas@rug.nl

Paris Avgeriou
p.avgeriou@rug.nl

Ronald Kruizinga
ronmatk@gmail.com

Ruben Scheedler
rubenscheedler@gmail.com

[1] Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, Faculty of Science and Engineering, University of Groningen, Nijenborgh 9, 9747AG Groningen, Netherlands

Published online: 16 December 2020

of poor design as they expose a *logical coupling* between the two files that is not explicitly declared in either of them [10]. Such a problematic design has impact on run-time qualities, such as Reliability, as co-changes are useful predictors of faults [17, 18, 38]. There exist several studies in the literature documenting this aspect of co-changes. Kim et al. [17], for instance, explain that when a programmer makes a change based on incomplete or incorrect knowledge, they likely cannot assess the impact of their modifications as well, thus introducing faults to nearby files, logically coupled to the changing file. Furthermore, when a file has a fault, there is a good chance that files that are "nearby" in the dependency network also contain a fault, and there is a good chance that they will change together when fixes are applied [17].

On the other hand, when analysing design-time qualities of a system (e.g. maintainability and evolvability), issues in the architecture of the system are among the most important and insightful to look at because of the critical role played by software architecture in shaping the system [19]. This has initiated a lot of research on how Technical debt [4] at the architecture level (i.e. Architectural debt, or ATD) negatively impacts Maintainability and Evolvability on the long term. One of the most interesting examples of ATD are architectural smells, which are defined as "[...]commonly (although not always intentionally) used architectural decision that negatively impact system quality" [13]. Architectural smells are a significant threat to the long-term sustainability of the system's architecture and hinder regular maintenance activities by increasing the complexity of the system.

Currently, the research community's interest on architectural smells is growing rapidly [42]. However, there are no studies looking at the interplay between architectural smells and co-changes; we note that there is instead mature research on code smells and co-changes, as well as antipatterns and co-changes [11, 29]. It is therefore interesting to study this relationship in order to better understand the intricacies and trade-offs between Reliability and Maintainability, through the two aforementioned indicators: co-changes and architectural smells, respectively.

This study makes a first step in this direction by setting up a case study to examine two important aspects: (1) how architectural smells and co-changes co-occur, and (2) which one precedes the other in appearing in a system.

The study focuses on dependency-based architectural smells, a category of smells for which there exist no other studies looking at their interconnection with co-changes. To conduct the study, we selected a set of 14 open source Java systems and mined their history in search of architectural smells and co-changing files. Next, we developed several hypotheses for each research question and tested them through statistical tests.

Our findings show that, on average, 50% of co-changing file pairs detected by our custom algorithm are also affected by at least one architectural smell. In addition, in seven projects, file pairs affected by an architectural smell were found to be more likely to co-change than non-affected pairs. In all projects, however, over 90% of all co-changes were detected before the smell. These findings allow us to understand better the interplay between Reliability and Maintainability throughout a system's evolution history.

The rest of the paper is organised as follows: "Related work" reports on similar studies from the literature; "Methodology" covers the case study design and the methods used to collect the data; "Architectural smells and co-changes (RQ1)", "Frequency of co-changes in smelly artefacts (RQ2)", and "Introduction order of co-changes and architectural smells (RQ3)" describe the methods used to analyse the data and report the results obtained for the three research questions of this study respectively; "Discussion" discusses the results; "Threats to validity" elaborates on the possible threats to the validity of this study; and finally "Conclusion" reports the concluding remarks of this paper.

Related Work

Architectural Smells

The literature contains several catalogues of architectural smells defined by a multitude of authors. In this section, we briefly mention some of these studies as well as key empirical studies on architectural smells.

Lippert and Roock [25] defined in 2006 a number of architectural smells that affect a system at different levels (class, package, module, etc.). Most of these smells were dependency-based smells, meaning that they were describing issues arising in the dependency network of a system, such as cyclic dependencies. Others were based on the size of the artefact affected, or on the inheritance hierarchy of a class.

In 2009, Garcia et al. [13] identified a four architectural smells defining suboptimal structures in how the functionality was implemented and distributed across the different parts of the system. The list was then further extended in 2012, by Macia et al. [22], who have also performed one of the first empirical analyses looking into the evolution of architectural smells over time. Their findings showed that code anomalies detected by the employed strategies were not related to architectural problems, highlighting how the tools used were neglecting the artefacts suffering from architectural problems.

Suryaranayana et al. [41] proposed in 2014 an extensive catalogue of design smells (some of which where similar to some architectural smell previously defined by other authors) identifying multiple categories: abstraction, modularisation, hierarchy, and encapsulation. The categories and the smells identified were all based on key object-oriented design principles.

Later on, Mo et al. [27] defined five new types of architectural smells in the context of the authors' research on Design Rule Spaces [43]. One type of smell is defined using the concept of logical coupling, identifying modules that, while do not directly depending upon each other, are not mutually independent and change together frequently.

Arcelli et al. [2, 12] provide a catalogue of three dependency-based architectural smell along with a validated tool to detect those smells in Java systems. More details on the smells defined by Arcelli et al. are reported in section 3.4.1.

Le Duc et al. [20, 21] strove to provide a formalised definition of AS before performing an empirical study on the evolution of the instances in 421 versions from 8 open source systems. They tested the hypotheses that (1) smelly files are more likely to have issues associated than clean files and that (2) smelly files are more likely to change than non-smelly ones, accepting them both.

Finally, in our previous study [36], we investigated the evolution of individual AS instances over time with respect to their characteristics, such as size, centrality, and age. Our findings showed that the vast majority of architectural smells instances tend to grow in size (number of elements affected, and/or number of connections among the affected elements) over time. Additionally, smells also tend to "move" towards the centre of the dependency network of the system, as measured by the Page Rank of the components affected by the smell. Another interesting finding showed that Cyclic Dependency instances were the less persistent type of smell in the 21 systems analysed, with only a 50% survival rate after 5 releases.

Co-changes

Jaafar et al. propose two types of co-changes: MCC (Macro Co-Changing) and DMCC (Dephase Macro Co-Changing) [14]. These concepts describe two files changing simultaneously (MCC) or nearly simultaneously (DMCC). Their approach, named Macocha, attempts to find files that are MCC or DMCC using a *sliding window*, splitting up the history of the project into periods of 5.17 hours and then defining a profile/vector that for each period contains whether the file has changed (1) or not (0), finally resulting in a binary string. These strings can be compared to find co-changes. If the strings match exactly, they are marked as DMCC. If they have a Hamming distance < 3, they are marked as MCC.

Bouktif et al. undertake another approach to mine co-changes, focusing on reducing computation time [9]. One typical problem with co-changes which they attempt to solve is that the examined window of time can influence the results. Taking a larger window of time means including (co-)changes that might no longer be relevant. Taking a smaller window might result in missing important change-sets, resulting in an excessive amount of possibly co-changing pairs. The authors find that larger windows result in better accuracy, but of course require more computation. They present Dynamic Time Warping (DTW) as an algorithm for finding co-changes, thereby solving the task in quadratic time respective to the length of the history (time window).

Zimmermann et al. [44] also look at mining co-changes using Market Basket Analysis (MBA). Every change-set is treated as a 'basket' containing several changes. Using the *apriori* algorithm they are able to mine association rules from histories of these change-sets. For a changed file, they are able to predict 26% of co-changed files. Moreover, 70% of the generated top three guesses turn out to be indeed co-changing.

Mondal et al. use MBA to mine co-changing method groups [28]. They analyse the change-sets of 7 open source projects and compare the lifetime and change-proneness of co-changing methods with those of non-co-changing methods. They found that co-changing methods indeed live longer and are more prone to change.

Co-changes are typically mined from VCS data, but Robbes et al. also try to find co-changes on a more fine-grained level [32]. They implemented extra software in the IDE of developers allowing them to see when changes occur within a development session. They constructed detailed metrics based on the amount of changes per file in a session and determined co-changes based on these. Although this approach provides more detailed data, it is also harder to collect this data. The collected data can also be dependent on the monitored developers. For this reason, in our paper we utilise 'traditional' VCS data.

SDK4ED Project

This work has been designed as part of the SDK4ED[1] (Software Development ToolKit for Energy Optimization and Technical Debt Elimination) project. The vision of SDK4ED is to minimize the cost, the development time and the complexity of low-energy software development processes, by providing tools for automatic optimization of multiple quality requirements, such as Technical debt, Energy efficiency, Dependability (i.e. Reliability, Availability, and Security) and Performance. One of the topics on which the project is concentrating its efforts the most is researching and developing tools to identify the trade-offs between runtime and

[1] Browse the project's website for more information: https://sdk4ed.eu/.

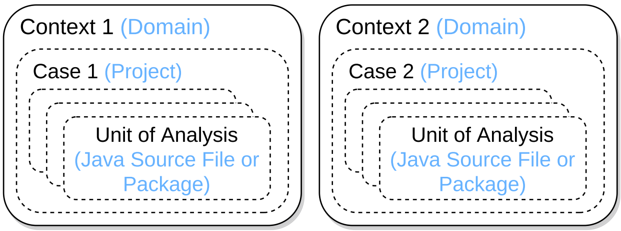

Context 1 (Domain)
Case 1 (Project)
Unit of Analysis (Java Source File or Package)
Context 2 (Domain)
Case 2 (Project)
Unit of Analysis (Java Source File or Package)

Fig. 1 Case study design representation. Based on Runeson et al.'s work [33]

design-time software quality attributes at multiple levels of abstractions (code, design, and architecture).

The project's research efforts so far have been focused on studying the trade-offs among different software quality attributes. For example, Papadopoulos et al. [30] have studied the interrelations between Maintainability-related metrics, Performance, and Energy consumption with a special focus on embedded systems. Their findings show that indeed there are trade-offs among these three qualities when refactorings and transformations are applied to improve one of the three qualities. The way SDK4ED deals with the interplay between quality attributes is described by Jankovic et al. [15].

Part of the authors of this paper also investigated trade-offs between quality attributes by analysing qualitative data collected from software architects and developers from seven different software companies Sas and Avgeriou [35]. The findings suggest that there are several trade-offs between quality attributes, but most of them are implicit, and the developers only realise they made a trade-off in hindsight. In this regard, the present study, instead, investigates quantitatively the interaction between Technical debt (i.e. Maintainability) and Reliability, a design-time and a runtime quality attribute, respectively.

Methodology

Case Study Design

This case study is set up according to the guidelines for case study design as described by Runeson et al. [33]. The general structure is displayed in Fig. 1. In this study, software projects function as cases and their packages and source files function as units of analysis. By analysing a multitude of projects, we are setting up a *multiple-case study*. Furthermore, since each case contains many different units of analysis, the study is an *embedded case study*.

We have chosen this setup to avoid bias in our results as software projects can vary in size, style and structure. Multiple-case studies allow to increase the chances of generalising the results to a greater population of projects, whereas *individual case-studies* do not offer this but have the advantage of gaining precise insights about the project under analysis.

Goal and Research Questions

The goal of this study is to understand the interplay between Reliability and Maintainability via two of their proxies. Using the Goal Question Metric approach [39], the goal can be formulated as follows:

> Analyse co-changes and architectural smells for the purpose of understanding the interplay between Reliability and Maintainability with respect to co-occurrence and moment of introduction from the point of view of software developers and architects in the context of open source Java software systems.

To ensure we study precisely what we state in our goal, we break it down into three research questions:

RQ1 *What is the overlap between co-changing artefacts and smelly artefacts?*

This is an exploratory RQ that investigates how exactly co-changes and architectural smells overlap. Specifically, we will investigate what fraction of the artefacts affected by smells happen to co-change, and vice versa. Understanding if co-changes, which are well known fault predictors [17, 18, 38], and architectural smells co-exist in the same components will shed some light on how Reliability and Maintainability issues are intertwined, and more precisely to what extent.

RQ2 *Are co-changes found more often in smelly artefacts?*

RQ2 follows up on RQ1, attempting to determine statistically whether smelly artefacts are in fact more prone to co-change than non-smelly ones. This can help us understand whether maintainability issues (in the form of architecture smells) drive the changes within the system: smelly artefacts could be hotspots where developers focus a lot of their efforts (possibly) due to their complexity and poor understandability.

RQ3 *Are smells introduced before or after files start co-changing?*

Table 1 Demographics of the projects selected for this study

Project	Description	Owner	Domain	KLOC Start-End
ArgoUML	UML modelling tool	Tigris-org	Documentation	78–145
Druid	Realtime analytics database	Apache	Databases	3–28
Jackson	JSON library	FasterXML	Formatted Data	34–57
JUnit5	Unit testing framework	JUnit-Team	Testing	1–20
MyBatis3	SQL object mapper	MyBatis	Databases	23–19
PDFBox	PDF manipulation	Apache	Formatted Data	47–63
POI	MS Office interaction	Apache	Formatted Data	70–94
PgJDBC	Postgresql Java Driver	Pgjdbc	Databases	8–28
Robolectric	Android unit testing	Robolectric	Testing	32–70
RxJava	Reactive JVM Extensions	ReactiveX	General purpose	11–143
Sonarlint	Linter for IntelliJ	SonarSource	General purpose	0–10
Swagger	API-documentation	Swagger	Documentation	0–15
TestNG	Testing framework	Cbeust	Testing	18–56
Xerces2	Java XML parser	Apache	Formatted data	62–118

Finally, with RQ3 we aim at investigating whether co-changes precede the appearance of architectural smells in the source code of the system, or it is the other way around, or maybe they are introduced simultaneously. This can reveal how the symptoms (i.e. co-changes and architectural smells) of poor design decisions arise within the system, which is crucial in understanding how these decisions affect the work of developers in the long term.

Case Selection

As mentioned above, software projects can differ from each other considerably. Analysing a wide variety of projects (cases) for our study is therefore important to increase external validity [33]. Following Runeson et al.'s guidelines, we opt to achieve the *maximal variance* in the distribution of the following properties of our cases:

– *Project size*: projects with a small, medium, and large amount of artefacts (or total lines of code).
– *Domain*: projects intended to be used in different domains and environments.
– *Owner*: projects with different owner(s), authors, and contributors.

To select the projects, we used (a) GitHub's most starred Java projects list[2], (b) Apache's projects list[3] and (c) projects used in previous empirical studies similar to the present work. To ensure there were enough changes in the repository and the development was still in progress, the projects were selected if they had at least 5 years of development with a minimum of 250 commits on the master branch and the last commit was in 2020. Additionally they also had to have at least 10 KLOC in the last commit analysed, to filter out toy projects and projects that would yield an excessively low number of co-changes and/or architectural smells.

The projects selected are reported in Table 1.

Data Collection and Tools

The data collection process was two-fold. First, we mined the architectural smells from the 14 projects selected for this study. To do so, we used Arcan [2] that detects architectural smells in the history of a system and AStracker [36] that tracks these smells from one version to the next. The architectural smells considered for this study are Cyclic Dependency (CD), Unstable Dependency (UD), and Hublike Dependency (HL). This set of smells was selected because it comprises some of the most important architectural smells to study, according to our current theoretical and empirical understanding [36].

The second step entails extracting the co-changes from the selected projects using two different algorithms: one existing algorithm (Dynamic Time Warping - DTW, see Sect. 2) that was used in previous studies on co-changes [9] and one custom algorithm, named Fuzzy Overlap (FO). Further details on each of these algorithms are presented in Sect. 3.4.2.

For each system analysed, we collected AS and co-change data points by analysing one commit a day for each day the project was changed since the beginning of its history in the Git repository.

[2] See https://github.com/search?l=&o=desc&q=language:java+pushed:>2020-01-01&s=stars&type=Repositories.

[3] See https://projects.apache.org/projects.html?language.

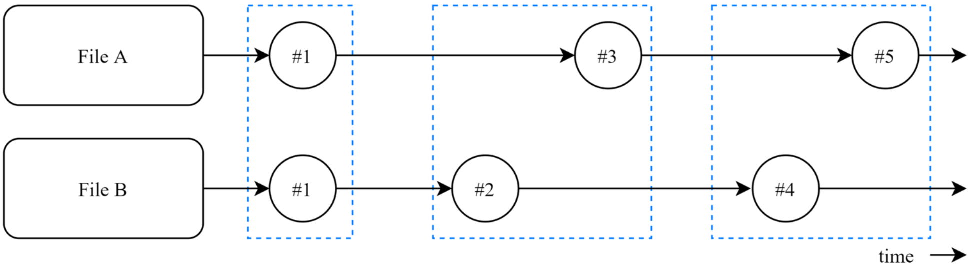


Fig. 2 The basic concept used by Fuzzy Overlap. The circles represent commits in which the files changed. In commit #1, both files change. After that, they do not change in the same commits anymore, but file B always changes just before file A

Architectural Smells

This section lists the architectural smells considered by this study. The definition of these smells is provided by Arcelli et al. [12] and briefly reported here.

Unstable Dependency This smell represents a package that depends upon a significant number of components that are less stable than itself, according to Martin's instabiliy metric [23], 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 main problem caused by UD is that the probability to change the central package grows higher as the number of unstable components it depends upon grows accordingly. This increases the likelihood that the components that depend upon it change as well when it is changed (ripple effect), thus inflating future maintenance efforts.

Hublike Dependency This smell represents a class or package 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 [12]. 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: outgoing dependencies are hard to change because several components (i.e. classes or packages) indirectly depend upon them; and incoming dependencies are more prone to changes caused by ripple effects propagated by the central component.

Cyclic Dependency This smell represents a cycle among a number of components; there are several software design principles that suggest avoiding creating such cycles [24, 25, 31, 40]. Cycles may have different topological shapes. Al-Mutawa et al. [1] have identified 7 of them. 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) [25]. In this study, we take into consideration both cycles between classes and cycles between packages.

Co-change Detection Algorithms

Dynamic Time Warping The dynamic time warping algorithm [34] is a way of measuring similarity between two time series, even if the speed of these time series varies. Traditionally, this algorithm has been used for automatic speech recognition, but it is also applied to a wide variety of other purposes, such as video, audio and graphics. It calculates the distance between two time series and provides a normalised version of the distance. If the distance is less than the threshold, we mark the corresponding file pair as co-changing. The threshold is set to 24 hours and is based on a case study performed by Bouktif et al. [9].

Fuzzy Overlap The fuzzy overlap algorithm is an algorithm that tries to formalise certain intuitive assumptions regarding co-changes in software development. These assumptions cannot be satisfied using more generic algorithms such as DTW. The algorithm, illustrated in Fig. 1 is based on the observation that co-changes can occur in a range of situations. They can occur either within the very same commit, when for example files A and B change at the same time, or there can be a short "delay" between the changes. For instance, if a change in File B is typically followed by a change in file A, as represented in Fig. 2, then a relationship between the two might exist and intuitively these two files would then be considered to be co-changing. Of course, if two files only change together once, this can easily be attributed to chance, instead of it being an actual co-change. In order to prevent this, FO implements a threshold for co-changes, filtering out all pairs that do not change together often enough. DTW is not capable of this distinction and will report every set of two files that change simultaneously as a co-change, as long as that change is their only change in that time period, as both will have identical change histories at that point.

The hyperparameters that FO algorithm uses to detect co-changes are (see Fig. 1):

- **Commit Distance**: the number of commits between two analysed commits. The value of this threshold was set based on the average number of commits in a day (excluding days without commits).
- **Time Distance**: the maximum time between two commits for them to be marked as co-changing. The value of this threshold was set using the third quartile of the

Table 2 Percentage of all file pairs reported as co-changing and their absolute value in parenthesis. Values over 5% are marked in bold

Project	% of files (number)		Total source code file pairs
	FO	DTW	
ArgoUML	4.48 (140,710)	0.55 (17,258)	3,140,960
Druid	3.73 (69,567)	2.05 (38,259)	1,866,807
Jackson	2.46 (3,474)	0.3 (497)	141,353
JUnit5	3.45 (11,506)	0.74 (2,477)	333,580
MyBatis-3	**38.22** (25,497)	0.19 (126)	66,703
PDFBox	0.76 (2,790)	0.12 (470)	368,982
PgJDBC	**12.25** (17,247)	0.17 (236)	140,824
POI	1.52 (11,404)	0.27 (2,029)	747,846
Robolectric	2.14 (41,071)	0.06 (1,236)	1,918,436
RxJava	3.24 (43,457)	*4.07* (54,644)	1,341,238
Sonarlint	3.58 (1,109)	0.26 (82)	30,987
Swagger	2.35 (1,395)	1.13 (673)	59,439
TestNG	2.85 (69,047)	***8.03*** (194,655)	2,425,206
Xerces2	3.37 (8,792)	2.19 (5,716)	260,670

interval time between commits, following the guidelines of Bird et al. [8].

- **Match Threshold**: the minimum number of overlapping commits of two files for them to marked as co-changing. This threshold was set by looking at the distribution of co-changes matches between files and selecting the 95th percentile for each project. The approach is based both on related research [8] and on our own experience with the data set.

An implementation of FO is freely available online.[4]

Comparison of the two algorithms Both algorithms were run on the same data set of co-changes; the number of pairs reported by each algorithm can be seen in Table 2. With the exception of four results (*MyBatis-3*, *PgJDBC*, and *TestNG* projects), all results were below 5% of all pairs.

In general, FO reported more co-changes than DTW did, except for the *RxJava* and *TestNG* projects. Aside from *PDFBox*, FO reported that more than 1% of all pairs co-change, whereas DTW only reported 6 projects above 1%.

Note that originally we also used another, very common co-change detection algorithm: Market Basket Analysis—MBA. However, using the configuration parameters suggested in the literature, we were not able to obtain a sufficient number of co-changes that would allow us to carry out our analysis for the vast majority of the projects. Therefore, we opted to exclude MBA from our results.

Architectural Smells and Co-changes (RQ1)

Methodology

To investigate **RQ1**, we will select, from our data set of co-changes, all the file pairs affected by at least one architectural smell. These pairs must match either one the following conditions:

Algorithm 1 The Fuzzy Overlap algorithm.

```
procedure FuzzyOverlap($gitRepo$)                     ▷ The co-changes in a git repository
    Let $changingCommits$ be a map of lists
    for each $commit$ in $gitRepo$ do
        for each $file$ in $commit$ do
            if $file$ changed in $commit$ then
                Add $commit$ to $changingCommits[file]$
            end if
        end for
    end for
    Let $cochange$ be a matrix
    for $i$ in $changingCommits$ do                                  ▷ $i$ is a file
        for $j$ in $changingCommits$ s.t. $j > i$ do           ▷ $j$ changes after $i$
            Calculate all pair of commits of the two files
            Filter commit pairs using commit distance
            Filter commit pairs using time distance
            if # matches > match threshold then
                $cochange[i, j]$ := True
            end if
        end for
    end for
    return $cochange$
end procedure
```

[4] See https://github.com/RonaldKruizinga/CoSmellingChanges.

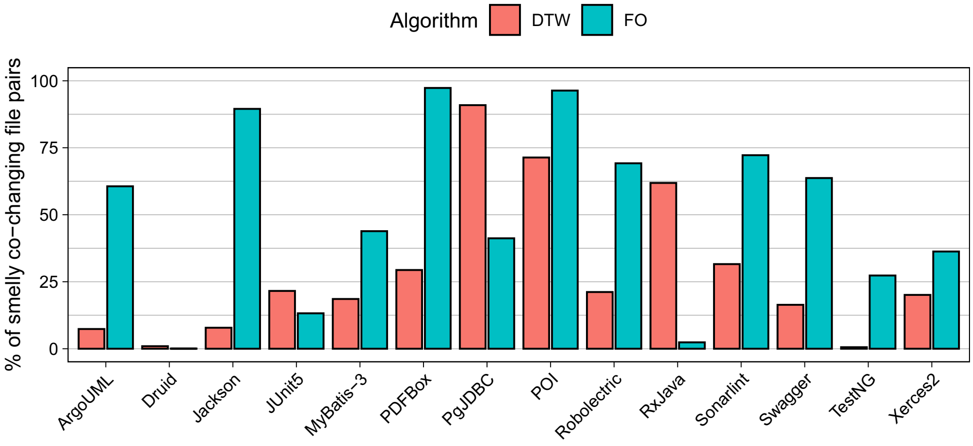


Fig. 3 Percentage of co-changing source file pairs that are smelly, by project and CC detection algorithm

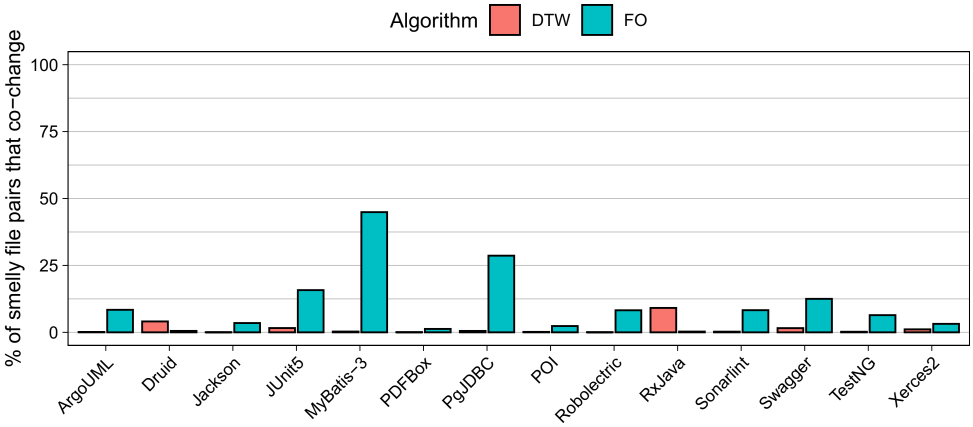


Fig. 4 Percentage of smelly source file pairs that are co-changing, by project and CC detection algorithm

Table 3 Comparison of the number of smelly and co-changing pairs divided by algorithm and the percentages (with the weighted values in parenthesis) of overlapping pairs w.r.t the total smelly and total co-changing pairs, respectively. Total source code file pairs: 9,674,544

Algorithm	No. of smelly pairs	% co-changing	No. of co-change pairs	% smelly
DTW	2,938,426	1.4 % (1.3%)	227,792	28.5 % (16.8 %)
FO		10.3 % (5.6%)	437,405	50.9 % (39.2 %)

$$\text{StartDate}_{\text{co}-change} \leq \text{EndDate}_{\text{smell}} \leq \text{EndDate}_{\text{co}-change} \quad (1)$$

$$\text{StartDate}_{\text{Smell}} \leq \text{EndDate}_{\text{co}-change} \leq \text{EndDate}_{\text{smell}} \quad (2)$$

Namely, the end date of a smell must be between the start and end date of a co-change, or vice versa. In other words, there must be at least some kind of overlap between the time periods a co-change and an architectural smell affected the same file pair.

RQ1 serves mostly as an exploratory question leading up to **RQ2**. It provides insight in where the most overlap is found between AS and co-changes.

Results

The results obtained from this research question are reported in Figs. 3 and 4, and in Table 3. In Fig. 3, we can note that the percentages of co-changing files that are also affected by an architectural smell reaches over 50% of the co-changing pairs as detected by FO in 7 projects. Lower percentages are instead detected when DTW is used to detect the co-changes and only 3 projects exhibit 50% or more of co-changing pairs affected by architectural smells.

On the other hand, Fig. 4 shows the percentages of smelly pairs that are also co-changing. In this case, only 2 projects exhibit smelly file pairs with more than 25% pairs that also co-change according to the FO algorithm. Given the sheer amount of smelly pairs, the DTW algorithm has practically

Table 4 Contingency table example for RQ2's χ^2 tests

	Co-changed	Not co-changed
No smell	x	z
Smell	w	y

detected very few co-changes in smelly files, with only 1% of the smelly files undergoing co-changes as detected by DTW (see Table 3). Instead, the FO algorithm was able to detect more, with an average of 10.3% of smelly file pairs also co-changing. To summarise, co-changing pairs, which represent **logically-coupled file pairs, are characterized by very high percentages of poor design** by taking part in architectural smells.

Frequency of Co-changes in Smelly Artefacts (RQ2)

Methodology

The answer to **RQ2** will be obtained through statistical analysis of two caretorigal variables. The first variable is whether a file pair is co-changing or not, and the second variable is whether two files belong to the same architectural smell. It is our aim to establish whether smelly artefacts are more likely to co-change than clean artefacts. Several statistical tests are applicable for this analysis, though the best candidates are either the χ^2 test for independence or the Fisher's exact test [37]. Based on the size of our data set, we opted for the χ^2-test. Fisher's test is best to be used with a sample size ≤ 20 [37]. Our data set is orders of magnitudes larger as our sample consists of all possible pairs of source code files in a repository (changed in the relevant time frame); thus Fisher's test would be unsuitable.

The input to the χ^2 test is a two by two contingency table containing the counts of observations with one of the four possible combinations of our variables. An example of such a contingency table can be found in Table 4.

Depending on the algorithm used to detect the co-changes, and on the scope (classes, packages, or both) of the architectural smells, we define multiple pairs of null and alternative hypotheses as follows:

- $H_0^{RQ2\_[algorithm]}$: Artefacts affected by AS are as likely to co-change as artefacts not affected by AS.
- $H_1^{RQ2\_[algorithm]}$: Artefacts affected by AS are more likely to co-change than artefacts not affected by AS.

Table 5 Results of testing H^{RQ2} with co-changes reported by FO and all AS

Project	$H_0^{RQ2\_FO}$	χ -value	p value	o	ϕ-value
ArgoUML	Rejected	**55067.14**	**<0.01**	**3.75**	**0.14**
Druid	Accepted	**399.77**	**<0.01**	0.10	0.02
Jackson	Accepted	**1133.84**	**<0.01**	**6.33**	0.09
JUnit5	Rejected	**4073.88**	**<0.01**	**5.65**	**0.11**
MyBatis-3	Rejected	**1237.40**	**<0.01**	**1.79**	**0.14**
PDFBox	Accepted	**1708.49**	**<0.01**	**18.53**	0.07
PgJDBC	Rejected	**4431.60**	**<0.01**	**3.61**	**0.18**
POI	Accepted	**5336.27**	**<0.01**	**12.37**	0.09
Robolectric	Rejected	**71237.67**	**<0.01**	**10.85**	**0.20**
RxJava	Accepted	**2833.66**	**<0.01**	0.20	0.06
Sonarlint	Rejected	**883.96**	**<0.01**	**6.11**	**0.17**
Swagger	Rejected	**2944.03**	**<0.01**	**12.44**	**0.24**
TestNG	Accepted	**12252.85**	**<0.01**	**2.65**	0.08
Xerces2	Accepted	**8.40**	**<0.01**	0.94	< .01

Bold font face indicates that the value satisfies the rejection criterion

Table 6 Results of testing H^{RQ2} with co-changes reported by DTW and all AS

Project	$H_0^{RQ3\_DTW}$	χ -value	p value	o	ϕ -value
ArgoUML	Accepted	**5655.45**	**<0.01**	.15	.04
Druid	Accepted	**42.54**	**<0.01**	**1.42**	< .01
Jackson	Accepted	**620.30**	**<0.01**	.05	.07
JUnit5	Accepted	**86.55**	**<0.01**	**2.19**	.02
MyBatis-3	Accepted	**28.94**	**<0.01**	**2.82**	.02
PDFBox	Accepted	**106.66**	**<0.01**	.34	.02
PgJDBC	Accepted	**121.88**	**<0.01**	**4.13**	.03
POI	Accepted	**340.64**	**<0.01**	.42	.02
Robolectric	Accepted	**41.38**	**<0.01**	.55	< .01
RxJava	**Rejected**	**26641.13**	**<0.01**	**5.76**	**0.17**
Sonarlint	Accepted	< .01	0.96	**1.02**	< .01
Swagger	Accepted	**6.83**	**<0.01**	**1.33**	0.01
TestNG	Accepted	**15129.81**	**<0.01**	.04	0.09
Xerces2	Accepted	**830.31**	**<0.01**	0.39	0.06

Bold font face indicates that the value satisfies the rejection criterion

RQ2 will be answered for both co-change detection algorithms and the respective null hypothesis for each test is denoted by the [*algorithm*] label.

Normally, one would reject H_0^{RQ2} when the test results in a χ-value > 3.84 (critical value) and a p-value < 0.05. However, since we are dealing with a considerable sample size, we will also calculate a corresponding effect size ϕ as defined by Eq. 3.

$$\phi = \sqrt{\frac{\chi^2}{n}} \qquad (3)$$

In Eq. 3, χ^2 is the value returned by our test and n is the sample size. The resulting value ϕ can take values in the interval $[-1, 1]$. The value indicates effect size in the following manner: $0.1 \le \phi < 0.3$ means a *small* effect, $0.3 \le \phi < 0.5$ means an *average* effect and $\phi \ge 0.5$ means a *large* effect [37]. To reject H_0^{RQ2}, the following must hold $\phi \ge 0.1$.

Moreover, to accept H_1^{RQ2}, we need to know the direction of the association our test might find, thus we calculate its odds ratio:

$$o = \frac{x * y}{w * z} \tag{4}$$

using the quantities listed in Table 4.

Results

The results obtained from testing the two null hypotheses for each project and for each algorithm are shown in Table 5 and in Table 6. By looking at Table 5, it can be noted that for 7 projects out of 14 in total (50 %) we reject the null hypothesis $H_0^{RQ2\_FO}$ for the FO algorithm. This means that for these projects, the artefacts affected by an AS are more likely to co-change than artefacts not affected by AS. We also note that 4 (28 %) more projects (Jackson, POI, PDFBox, and TestNG) were close to the required ϕ-value threshold and passed the remaining three conditions.

Table 6, shows the results obtained using the co-changes detected by the DTW algorithm. In this case, we reject the null hypothesis $H_0^{RQ2\_DTW}$ for 1 project out of 14 in total (7%), meaning that in the vast majority of the projects, the co-changes detected by DTW are as likely to appear in smelly artefacts as in non-smelly ones. Unlike for the FO algorithm, in this case, the 6 (42%) projects that passed the first three conditions were not close to passing the ϕ-value threshold.

Given these results, we accept the null hypothesis $H_0^{RQ2\_DTW}$ for the DTW algorithm as there is not sufficient evidence to reject it. For the FO algorithm, given the results and the very strict criteria, we conclude that although there is not enough evidence to reject the null hypothesis $H_0^{RQ2\_FO}$ categorically, there is instead enough evidence to affirm that, in *most projects*, smelly file pairs are more prone to co-change than non-smelly ones.

Introduction Order of Co-changes and Architectural Smells (RQ3)

Methodology

Answering **RQ3** requires to determine when a pair of smelly source code files has started co-changing and when the smell affecting them was introduced. After determining this information, we partition our data set into three groups:

1. $\text{Emergence}_{\text{smell}} < \text{Emergence}_{\text{co}-change}$ *(smell-earlier)*
2. $\text{Emergence}_{\text{smell}} > \text{Emergence}_{\text{co}-change}$ *(co-change-earlier)*
3. $\text{Emergence}_{\text{smell}} = \text{Emergence}_{\text{co}-change}$ *(simultaneous)*

where $\text{Emergence}_{\text{smell}}$ is the date of the commit in which the smell is introduced and $\text{Emergence}_{\text{co}-change}$ is the date of the first commit in which both files of the co-change changed. The simultaneous group, however, ends up having a relatively low number of pairs (statistically insignificant), and therefore we opt to ignore it for the rest of this sub-section for the sake of brevity (we do show the results for this group in the next sub-section). Obviously, co-changes and smells that have no overlap are also left out of this analysis.

The two remaining partitions can be seen as a binomial distribution, where either one of the following two events can occur: *success*, where one phenomenon indeed precedes the other, or *failure*, for which this is not true. The binomial distribution implies that RQ3 can be answered using the binomial sign test [37].

For the null hypothesis, the expected balance between the two outcomes is 1 to 1. In other words, it is expected that in 50% of overlapping pairs the smell is introduced first and in the other 50% the co-change comes first.

Let π_1 be the probability of a pair falling in category 1, and let π_2 be the probability of it falling into category 2 such that $\pi_1 + \pi_2 = 1$. A null hypothesis can then be formed based on the expected value for π_1. This value is set to .5, capturing the equal distribution of earlier co-changes and earlier smells.

We are not merely interested in whether the distribution of earlier co-changes and smells matches the expected one, but also whether the *skewing direction* is a match. Therefore, two one-tailed tests are used instead of one two-tailed test. This gives rise to the following hypotheses:

a. Are smells introduced before files start co-changing?

- $H_0^{RQ3a\_[algorithm]} : \pi_s \le 0.5$
- $H_1^{RQ3a\_[algorithm]} : \pi_s > 0.5$

b. Are co-changes introduced after files start smelling?

- $H_0^{RQ3b\_[algorithm]} : \pi_c \le 0.5$
- $H_1^{RQ3b\_[algorithm]} : \pi_c > 0.5$

where π_s is the probability of a smell occurring before a co-change and π_c the probability of the co-change coming first. Note that the null hypotheses include $\pi_s < 0.5$. This is explained in the next paragraph. The analyses will be

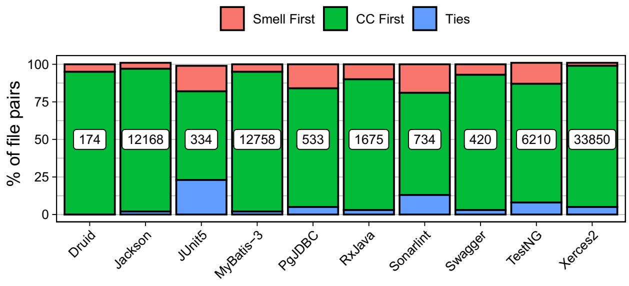


Fig. 5 Introduction order of smelly co-changing pairs in percentage w.r.t. the total number (shown at the centre of each bar) for the FO algorithm

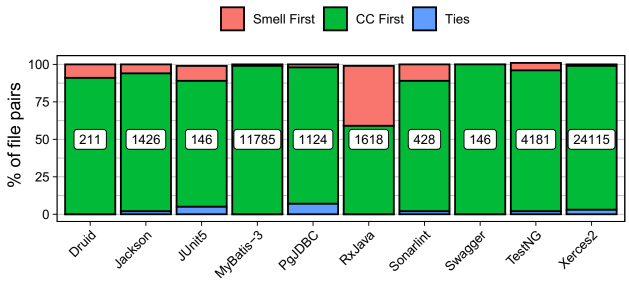


Fig. 6 Introduction order of smelly co-changing pairs in percentage w.r.t. the total number (shown at the centre of each bar) for the DTW algorithm

performed in twofold, namely for the reported overlapping pairs of FO and DTW (represented by *[algorithm]* in the hypotheses). With respect to the smells that are considered, both package-level and class-level smells are included.

The null hypotheses are rejected when two conditions are met. Firstly, earlier smells and earlier co-changes must occur more often. Secondly, the probability of the observed amount of successes (p-value) *or more* must be lower than .05. Say, for example, that m smells occurred earlier and n co-changes. H_0^{RQa} may then be rejected when the probability (p-value) of observing m or more smell-earlier pairs is lower than confidence level $\alpha = 0.05$. This comes down to calculating the cumulative probability of observing *m, m+1, ...* up to *m+n* smell-earlier pairs. When only the p-value is evaluated, the direction of skewing remains unknown, and this would correspond with a null hypothesis of the form $\pi \neq 0.5$. The extra condition validates the direction and means that either H_1^{RQ3a} or H_1^{RQ3b} can be accepted.

Results

Before enunciating the results, we would like to note that, due to memory constraints, we were not able to calculate all the necessary data to answer RQ4 for ArgoUML, PDFBox, POI, and Robolectric.

For the other projects, Figs. 5 and 6 depict the number of file pairs that were smelly before they started co-changing, or vice versa, for the two algorithms FO and DTW, respectively. Ties are also shown for completeness and represent a low percentage of the total cases. We observe that co-changes consistently appear before an architectural smell is introduced in the same file pair. This is valid for all projects and both algorithms.

The statistical tests return the exact same result: H_0^{RQ3a} is accepted and H_0^{RQ3b} is rejected for all the projects and both algorithms. We therefore conclude that file pairs start co-changing before a smell starts affecting that same file pair, meaning that co-changes precede architectural smells.

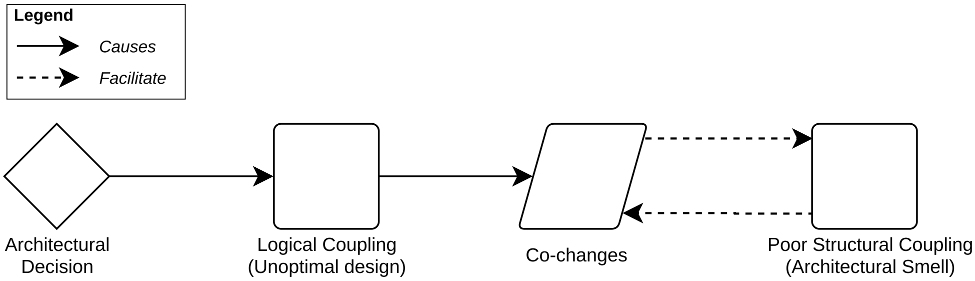
Discussion

The results from RQ1 allow us to explore the overlap between architectural smells and co-changes. Looking at Fig. 3, it is interesting to note that several projects have a remarkably high percentage of co-changing pairs (from either algorithm) that are smelly. This confirms that logical coupling is a sign of poor architecture and has adverse effects on system quality (in the form of architectural smells).

A very different result is illustrated in Fig. 4 regarding the percentage of smelly file pairs that also co-change for each project. Such percentages are relatively low because because most smells affect more than two components [36], like for example a cycle affecting 10 elements. The files that take part in this cycle that have direct dependencies are more likely to co-change than a random pair of files from the same smell without a direct dependency connecting them. A factor influencing this is the way change propagation probability (due to ripple effects) diminishes the “farther” a file is (in the dependency network of the system) from the changing file [3]. Additionally, we only consider overlaps with smelly pairs from the same smell. Co-changing pairs that are affected by two different smells are not considered in this study.

Another interesting finding from this research question is the difference in the co-changes detected by the two algorithms considered (see Fig. 3). The co-changes detected by FO seem to be more correlated to the presence of AS than DTW’s. A possible explanation is the fact that DTW was configured with parameters from the state of the art, calibrated based on two projects only [9], whereas FO was configured by rigorously selecting each hyperparameter based on a statistical analysis of each project’s commit frequency. Hence, from this point of view, one could argue that FO is a better co-change detection algorithm because it is able to find co-changes in files that manifest structural issues better than DTW.

Fig. 7 Architectural smells' introduction in a system induced by logical coupling


Legend
Causes
Facilitate
Architectural Decision
Logical Coupling (Unoptimal design)
Co-changes
Poor Structural Coupling (Architectural Smell)

The findings from RQ2 highlight that smelly files are more likely to co-change (as detected by the FO algorithm) than non-smelly files. The main implication of this finding is that components affected by architectural smells may be burdened with extra maintenance effort, increasing the technical debt interest paid by developers. The higher proportion of co-changing artefacts in smelly components means that architectural smells *indirectly affect* the level of Reliability of the affected components, as co-changes are found to be predictors of faults [17, 18, 38]. Architectural smells are not the only type of problem that has been found to increase the change-proneness of the affected components, in fact, components affected by code smells and antipatterns were found to have an increased change- and fault-proneness too [16]. Therefore, low Maintainability levels at different levels of abstraction (code, design, and architecture) may negatively impact Reliability because low quality components require more frequent changes by the developers, increasing the chances of eventually introducing faults.

RQ3 shows that in over 90% of the file pairs where an overlap between co-changes and an architectural smells occurs, the co-change precedes the architectural smell. This is a very interesting finding that shows that, eventually, up to 50% of the files that consistently change together (see Table 2) end up manifesting maintainability issues (architectural smells). We conjecture that this is to some extent caused as a consequence of the co-changing process itself: in order to fix the issues arising (or adapt the system to the new requirements) in the co-changing files, new code is added, new dependencies are introduced, and the original dependency structure of the two files grows more complicated, resulting in the introduction of architectural smells as the original design of the system is eroded. In our previous work [36] we studied the evolution of architectural smell instances over time and discovered that architectural smells are a by-product of the software development process, since they are continuously introduced as the system grows in size (i.e. total lines of code). Indeed, the findings of this study corroborate that co-changing files are one of the possible factors leading up to the introduction of smells as the size of a system increases.

It is also possible that this process, especially when time is of critical concern, could create a vicious circle of changes: poor design introduces logical coupling within the entities of the system, allowing co-changes to arise, which increase the risk of introducing faults [17]. Fixing faults, however, causes logically-coupled files to be changed together [17], which may increase the chances that new smells are introduced (RQ3 results) by means of new code and dependencies. The unhealthy dependency structure that characterises architectural smells increases the chances that (co-)changes become even more frequent due to the presence of structural links between the affected elements [3] (ripple effects). Evidence of a similar process (i.e. cause-effect loops) were also found by Martini et al. [26] in their study on ATD items and their causes. This process is part of the larger process of architectural erosion that every system goes through as it ages [6, 19].

Another interesting point of discussion stemming from our results is how co-changes and architectural smells become intertwined. According to Garcia et al. [13], architectural smells are "***commonly** (although not always intentionally)* ***used** architectural **decisions*** [...]". Our results point towards a bigger picture: poor architectural decisions cause logical coupling, which in turn causes co-changes to arise because the concerns were not properly separated among the entities involved in the decision (see Fig. 7). Subsequently, the logical coupling among the entities creates the conditions for the smell to manifest itself in the dependency network of the system as actual (structural) dependencies. The affected component is now both logically and structurally coupled: changes are even more likely to propagate, initiating and propagating in the vicious circle mentioned in the previous paragraph. This expands our understanding of what an architectural smell is: it does not simply manifest a poor architecture decision but rather it represents the visible ramifications caused by that decision (e.g. a cycle in the dependency network). There are other (structurally) invisible ramifications like logical coupling and co-changes.

However, not all smelly artefacts co-change; in fact only 10.3% of smelly pairs co-change (see Table 3), implying that the remaining 89.7% of the smells could appear either directly (the design is inherently flawed), or perhaps through other processes similar to the one just described, as part of the larger process of architectural erosion.

Threats to validity

In this section, the limitations and threats to validity of the study are discussed as described by Runeson et al. [33] in terms of *construct validity*, *external validity* and *reliability*. As we did not look at causal relationships, *internal validity* is not relevant to this study [33].

Construct Validity

Construct validity reflects to what extent the study measures what it claims to be measuring and what is being investigated according to the research questions. To ensure construct validity, we adopted the case study design guidelines by Runeson et al., and improved the study in iterations during the process. This way, the data collection and analysis was planned out in advance in order to closely match the research questions. Nevertheless, we did identify a number of threats to construct validity.

The first threat are the start and end dates of a co-change. These dates are set to the first and last moment when the pair co-changes. However, this ignores the content of these changes and the distances between co-changes. Due to this, the date ranges can easily become enormous, possibly skewing the results. This was partially mitigated by the threshold percentile of the FO algorithm which filtered out file pairs that did not change often enough (Match threshold).

The second threat to validity is that there was little to no overlap between the co-changes detected by the two algorithms in the majority of the projects, in other words, the two algorithms returned rather different co-changes. This might have been caused by the fact that DTW uses a fixed threshold for all projects, whereas FO uses project-specific adaptive thresholds. To ensure the two algorithms were performing correctly, we carefully selected the thresholds using techniques and values from the state of the art. For the DTW algorithm, we selected the threshold based on Bouktif et al.'s work [9], who performed a case study on two projects and identified a threshold using different metrics. For the FO algorithm, instead, we calibrated the thresholds using the guidelines on analysing historical software data suggested by Bird et al. [8].

External Validity

External validity is concerned with how well the results of this study can be extended to other projects with a similar context [33]. A few possible threats can be identified.

The first involves the choice of projects. All are open source projects, which means that the results can only be generalised to other open source projects, and not necessarily to other kinds of projects. In addition, 5 out of 14 projects are owned by the Apache Foundation, which impacts the generalisation of results to other organisations. We have, however, made sure to mitigate this by choosing projects from 5 different domains, each with a similar number of projects.

The second threat is regarding the specific architectural smells that were chosen to analyse. It is incredibly difficult, if not impossible, to generalise the results unto other architectural smells as the results greatly depend on the type of smell and its detection strategy.

Reliability

Reliability is concerned with the extent to which the data collected and the analysis performed are dependent on the specific researchers.

All tools and scripts used for this study are freely available. This allows researchers to replicate results using the same data and parameters, and to run the same analysis on a different set of projects. Intermediate findings and data analysis steps were inspected and regularly discussed by the authors in order to ensure reliability.

In addition, similar data collection and analysis techniques have been used in previous studies on architectural smells [36] and co-change detection [7–9], assuring that such an approach to the analysis of these artefacts is possible.

Conclusion

This study has investigated co-changes and their relation to architectural smells (AS), as proxies of reliability and maintainability. A case study was set up analysing 14 open source projects and an accumulated 20,000 change-sets (commits), capturing decades of software change history and architectural smell instances. Two algorithms were then used to detect the co-changes, which we then merged with the architectural smell data to create our data set.

The data set was then explored and statistically analysed from several perspectives. The results have shown that 50% of co-changes detected by FO eventually become smelly artefacts. Moreover, in 50% of the projects, artefacts affected by AS were more likely to co-change than artefacts not affected, indicating that AS increases maintenance effort in certain projects and eventually impacting the Reliability of the affected components. The co-changes detected by both algorithms were also found to precede smells in over 90% of the cases, implying (along with the results obtained from RQ1) that some co-changes are early symptoms of architectural problems that have yet to manifest themselves in the source code of the system.

In conclusion, this work has provided key insights on the interplay between Reliability and Maintainability, using co-changes and architectural smells as proxies for these two qualities, respectively, and highlighting how low Maintainability negatively impacts Reliability.

Acknowledgements We would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster. 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. Al-Mutawa HA, Dietrich J, Marsland S, McCartin C. On the shape of circular dependencies in java programs. In: Proceedings of the Australian Software Engineering Conference, ASWEC, IEEE, pp 48–57; 2014. https://doi.org/10.1109/ASWEC.2014.15. http://ieeexplore.ieee.org/document/6824106/. Accessed 08 Dec 2020.
2. Arcelli Fontana F, Pigazzini I, Roveda R, Tamburri D, Zanoni M, Nitto ED. Arcan: a tool for architectural smells detection. In: Proceedings—2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings pp 282–285; 2017. https://doi.org/10.1109/ICSAW.2017.16
3. Arvanitou EM, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. A method for assessing class change proneness. In: ACM International Conference Proceeding Series, Association for Computing Machinery, vol Part. 2017;F1286:186–95. https://doi.org/10.1145/3084226.3084239. Accessed 08 Dec 2020.
4. Avgeriou P, Kruchten P, Ozkaya I, Seaman C. Managing technical debt in software engineering (Dagstuhl Seminar 16162). Dagstuhl Rep. 2016;6(4):110–38. https://doi.org/10.4230/DagRep.6.4.110.
5. Barney S, Petersen K, Svahnberg M, Aurum A, Barney H. Software quality trade-offs: a systematic map. Inform Softw Technol 2012;54(7):651–62. https://doi.org/10.1016/j.infsof.2012.01.008.
6. Bass L, Clements P, Kazman P. Software Architecture in Practice, 3rd edn. Addison-Wesley Professional; 2012. https://dl.acm.org/citation.cfm?id=2392670. Accessed 08 Dec 2020.
7. Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A. An empirical study on the developers' perception of software coupling. In: 2013 35th International Conference on Software Engineering (ICSE), pp 692–701 (2013)
8. Bird C, Menzies T, Zimmermann T. The art and science of analyzing software data. Burlington: Morgan Kaufmann; 2015.
9. Bouktif A, Gueheneuc Y, Antoniol G. Extracting change-patterns from CVS repositories. In: 2006 13th Working Conference on Reverse Engineering, 2006;221–230, https://doi.org/10.1109/WCRE.2006.27
10. D'Ambros M, Lanza M, Lungu M. Visualizing co-change information with the evolution radar. IEEE Trans Softw Eng. 2009;35(5):720–35. https://doi.org/10.1109/TSE.2009.17.
11. de Oliveira MC, Freitas D, Bonifácio R, Pinto G, Lo D. Finding needles in a haystack: leveraging co-change dependencies to recommend refactorings. J Syst Softw. 2019;158:110420. https://doi.org/10.1016/j.jss.2019.110420.
12. Fontana FA, Pigazzini I, Roveda R, Zanoni M. Automatic detection of instability architectural smells. In: Proceedings—2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016 2016;pp 433–437. https://doi.org/10.1109/ICSME.2016.33. Accessed 08 Dec 2020.
13. Garcia J, Daniel P, Edwards G, Medvidovic N. Dentifying Architectural Bad Smells. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR. 2009; pp 255–258 https://doi.org/10.1109/CSMR.2009.59
14. Jaafar F, Gueheneuc Y, Hamel S, Antoniol G. An exploratory study of macro co-changes. In: 2011 18th Working Conference on Reverse Engineering. 2011;325–334.
15. Jankovic M, Kehagias D, Siavvas M, Tsoukalas D, Chatzigeorgiou A. The sdk4ed approach to software quality optimization and interplay calculation.2019. https://doi.org/10.13140/RG.2.2.31377.58723.
16. Khomh F, Penta MD, Guéhéneuc YG, Antoniol G. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Emp Softw Eng. 2012;17(3):243–75. https://doi.org/10.1007/s10664-011-9171-y.
17. Kim S, Zimmermann T, Whitehead EJ, Zeller A. Predicting faults from cached history. In: Proceedings - International Conference on Software Engineering. 2007;489–498 https://doi.org/10.1109/ICSE.2007.66.
18. Kouroshfar E. Studying the effect of co-change dispersion on software quality. In: Proceedings—International Conference on Software Engineering. 2013;1450–1452 https://doi.org/10.1109/ICSE.2013.6606741.
19. Kruchten P, Nord RL, Ozkaya I. Technical debt: from metaphor to theory and practice. IEEE Softw 2012;29(6):18–21. https://doi.org/10.1109/MS.2012.167
20. Le DM, Carrillo C, Capilla R, Medvidovic N (2016) Relating architectural decay and sustainability of software systems. In: Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, IEEE, pp 178–181, doi:10.1109/WICSA.2016.15, http://ieeexplore.ieee.org/document/6824106/0
21. Le DM, Link D, Shahbazian A, Medvidovic N. An empirical study of architectural decay in open-source software. In: Proceedings—2018 IEEE 15th International Conference on Software Architecture, ICSA 2018, IEEE. 2018; 176–185 https://doi.org/10.1109/ICSA.2018.00027, https://ieeexplore.ieee.org/document/8417151/. Accessed 08 Dec 2020
22. Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A.Are automatically-detected code anomalies relevant to architectural modularity? In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development—AOSD '12. 2012;167 https://doi.org/10.1145/2162049.2162069, http://dl.acm.org/citation.cfm?doid=2162049.2162069
23. Martin RC. OO Design Quality Metrics. Qual Eng. 1994;8(4):537–42. https://doi.org/10.1080/08982119608904663.
24. Martin RC (2000) Design principles and design patterns. Object Mentor
25. Martin Lippert SR. Refactoring in large software projects: performing complex restructurings successfully. Wiley; 2006. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470858923.html. Accessed 08 Dec 2020.

26. Martini A, Bosch J. The danger of architectural technical debt: contagious debt and vicious circles. In: Proceedings—12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015. 2015;1–10 https://doi.org/10.1109/WICSA.2015.31
27. Mo R, Cai Y, Kazman R, Xiao L. Hotspot patterns: the formal definition and automatic detection of architecture smells. In: Proceedings—12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015. 2015;51–60 https://doi.org/10.1109/WICSA.2015.12
28. Mondal M, Roy CK, Schneider KA. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In: 2013 21st International Conference on Program Comprehension (ICPC). 2013;103–112
29. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting bad smells in source code using change history information. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2013;268–278
30. Papadopoulos L, Marantos C, Digkas G, Ampatzoglou A, Chatzigeorgiou A, Soudris D. Interrelations between software quality metrics, performance and energy consumption in embedded applications. In: Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, Association for Computing Machinery, New York, NY, USA, SCOPES '18, p 62-65 (2018) https://doi.org/10.1145/3207719.3207736
31. Parnas DL. Designing software for ease of extension and contraction. IEEE Trans Softw Eng. 1979;2:128–38.
32. Robbes R, Pollet D, Lanza M. Logical coupling based on fine-grained change information. In: 2008 15th Working Conference on Reverse Engineering, pp 42–46 (2008) https://doi.org/10.1109/WCRE.2008.47
33. Runeson P, Höst M, Rainer A, Regnell B. Case study research in software engineering—guidelines and examples. 1st ed. Hoboken: Wiley; 2012.
34. Sakoe H, Chiba S. Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans Acoust Speech Signal Process. 1978;26(1):43–9.
35. Sas D, Avgeriou P. Quality attribute trade-offs in the embedded systems industry: an exploratory case study. Softw Qual J. 2019. https://doi.org/10.1007/s11219-019-09478-x.
36. Sas D, Avgeriou P, Arcelli Fontana F. Investigating instability architectural smells evolution: an exploratory case study. In: 35th International Conference on Software Maintenance and Evolution, IEEE. 2019; 557–567. https://doi.org/10.1109/ICSME.2019.00090. https://ieeexplore.ieee.org/document/8919109/. Accessed 08 Dec 2020.
37. Sheskin DJ. Handbook of Parametric and Nonparametric Statistical Procedures, 5th edn. Chapman & Hall/CRC. (2007) https://doi.org/10.5555/1529939
38. Shihab E, Mockus A, Kamei Y, Adams B, Hassan AE. High-impact defects: a study of breakage and surprise defects. In: SIGSOFT/FSE 2011—Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press, New York, New York, USA, pp 300–310 (2011) https://doi.org/10.1145/2025113.2025155
39. van Solingen R, Basili V, Caldiera G, Rombach HD. Goal question metric (GQM) approach. Encycloped Softw Eng. 2002. https://doi.org/10.1002/0471028959.sof142.
40. Stevens WP, Myers GJ, Constantine LL. Structured design. IBM Syst J. 1974;13(2):115–39.
41. Suryanarayana G, Samarthyam G, Sharma T. Refactoring for software design smells: managing technical debt. Burlington: Morgan Kaufmann; 2014.
42. Verdecchia R, Malavolta I, Lago P. Architectural technical debt identification: the research landscape. In: 2018 ACM/IEEE International Conference on Technical Debt; 2018. https://doi.org/10.1145/3194164.3194176. http://www.ivanomalavolta.com/files/papers/TechDebt_2018.pdf. Accessed 08 Dec 2020.
43. Xiao L, Cai Y, Kazman R. Design rule spaces: a new form of architecture insight. In: Proceedings of the 36th International Conference on Software Engineering—ICSE 2014, ACM Press, New York, New York, USA 2014; 967–977 https://doi.org/10.1145/2568225.2568241
44. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, USA, ICSE '04. 2004; 563–572