university of groningen

faculty of science and engineering

UNIVERSITY OF GRONINGEN

MASTER'S THESIS

Exploring the Relation between Co-changes and Architectural Smells

---

Authors:
Ronald KRUIZINGA,
Ruben SCHEEDLER

Supervisors:
Prof. dr. ir. Paris AVGERIOU,
Darius SAS

A thesis submitted in fulfillment of the requirements for the Master's Degree in Computing Science

in the

Faculty of Science and Engineering

Department of Computing Science

July 17, 2020

Exploring the Relation between Co-changes and Architectural Smells

by Ronald KRUIZINGA, Ruben SCHEEDLER

In the last decade, the technical debt metaphor has gradually grown in popularity and is now become the preferred way for both practitioners and researchers to discuss the effort, costs and issues arising during software development activities. Detecting technical debt is one of the first steps to limit its growth and eventually pay it back. Co-changes are artefacts that over time change in a similar way and are an indicator of technical debt. Architectural smells are combinations of architectural decisions that reduce system maintainability, and are a form of technical debt.
The goal of this thesis is twofold, namely to investigate the possible relationship between co-changes and architectural smells and to compare different ways of mining co-changes. If co-changes are related to architectural smells, detecting co-changes can be used to trace technical debt.

To this end, this research introduces a novel way of detecting co-changes called "Fuzzy Overlap". This approach is compared with state-of-the-art approaches such as Market Basket Analysis and Dynamic Time Warping.

Regarding the relation between co-changes and architectural smells, this study attempts to analyze its direction, whether co-changes are more often found in smelly artefacts, and whether the smells are introduced before or after file pairs start to co-change.

From this analysis, it has become clear that the output produced by the Fuzzy Overlap algorithm tends to vastly differ from that generated by Market Basket Analysis and Dynamic Time Warping. For 50% of the projects analyzed, a relation between architectural smells and co-changes was found and for 100% of the projects it was found that co-changing precedes the introduction of architectural smells.

Acknowledgments

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. We would like to thank Darius Sas as our supervisor for the large amount of advice, tips and feedback given for the duration of the project.

Contents

Abstract i

Acknowledgments ii

Contents iii

List of Figures vi

List of Tables viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Technical Debt . . . 1
- 1.2 Software systems evolution . . . 2
  - 1.2.1 Version Control Systems . . . 2
  - 1.2.2 Co-changes . . . 3
- 1.3 Goals & Motivation . . . 3
- 1.4 Approach . . . 4
- 1.5 Outline . . . 4

2 Related Work 5
- 2.1 Background . . . 5
- 2.2 VCS . . . 6
- 2.3 Co-changes . . . 7
- 2.4 Architectural Smells . . . 8

3 Research Methodology 10
- 3.1 Case Study Design . . . 10
- 3.2 Goal and Research Questions . . . 11
- 3.3 Analyzed Projects . . . 11
- 3.4 Approaches to detect co-changes . . . 12
  - 3.4.1 Fuzzy Overlap . . . 12
    - Hyperparameters . . . 15
    - Commit Distance . . . 16
    - Time Distance . . . 16
    - Time Range . . . 17
    - Match Threshold . . . 17
  - 3.4.2 Market Basket Analysis . . . 18
  - 3.4.3 Dynamic Time Warping . . . 18

3.5 Data Collection . . . 19
3.5.1 Architectural smells . . . 19
Smell-scopes . . . 19
Smell-affected pairs . . . 19
3.5.2 File Changes . . . 20
3.5.3 Co-changes . . . 20
Fuzzy Overlap . . . 20
Market Basket Analysis . . . 20
Dynamic Time Warping . . . 20
3.5.4 Overlapping pairs . . . 21

4 Data Analysis **22**
4.1 RQ1 - What are the differences between the three co-change detection algorithms FO, MBA, DTW? . . . 22
4.2 RQ2 - Are artefacts affected by architectural smells co-changing? . 22
4.3 RQ3 - Are co-changes more often found in smelly artefacts? 23
4.4 RQ4 - Are smells introduced before or after files start co-changing? 24

5 Results **27**
5.1 RQ1 - What are the differences between the three co-change detection algorithms FO, MBA, DTW? . . . 27
5.1.1 Overlap . . . 28
5.1.2 Co-changes over time . . . 29
5.2 RQ2 - Are artefacts affected by architectural smells co-changing? . 30
5.2.1 Results for Class AS . . . 30
FO . . . 30
DTW . . . 31
5.2.2 Results for Package AS . . . 32
FO . . . 32
DTW . . . 32
5.2.3 Results for Class and Package AS . . . 33
FO . . . 33
DTW . . . 33
5.3 RQ3 - Are co-changes more often found in smelly artefacts? 34
5.3.1 Result for Class AS . . . 34
FO . . . 34
DTW . . . 36
MBA . . . 36
5.3.2 Result for Package AS . . . 37
FO . . . 37
DTW . . . 38
MBA . . . 38
5.3.3 Result for Class and Package AS . . . 39
FO . . . 39
DTW . . . 40
MBA . . . 40
5.3.4 Summary . . . 40
5.4 RQ4 - Are smells introduced before or after files start co-changing? 41

6 Discussion **44**

7 Study Limitations **47**
7.1 Construct Validity 47
7.2 External Validity 48
7.3 Reliability 49

8 Conclusion **50**
8.1 Future Work 51

A RQ4 Results **52**
A.1 RQ4a Results 52
A.1.1 FO 52
A.1.2 DTW 52
A.1.3 MBA 53
A.2 RQ4b Results 53
A.2.1 FO 53
A.2.2 DTW 53
A.2.3 MBA 54

B Hyperparameter Analysis Results **55**

Bibliography **56**

List of Figures

2.1 Mapping between changeset histories of two files [45]. 6

3.1 Setup of the case study using examples. 10
3.2 The basic idea of fuzzy mapping. 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, but file B always changes just before file A. 13
3.3 This figure compares two typical file history types. Files B and C (orange) are spotted by DTW as co-changing as their history is identical. However, the *support* of the similarity is minimal. Files A and D are less likely to be matched by DTW as their paths differ; however, FO will pick up such file pairs. 13
3.4 Flowchart of the FO algorithm. 15
3.5 Difference between commit distance and time distance hyperparameters used by FO. 16
3.6 Distribution of observed match counts among co-change candidate pairs. A threshold of 12 was selected for this dataset. 17
3.7 The two conditions for an overlapping pair (OP). 21

5.1 Overlap of output of the algorithm on three projects. 28
5.2 Co-changes over time. 29
5.3 File pairs affected by class-level smells also reported as co-changing by FO. 30
5.4 File pairs affected by class-level smells also reported as co-changing by DTW. 31
5.5 File pairs affected by package-level smells also reported as co-changing by FO. 32
5.6 File pairs affected by package-level smells also reported as co-changing by DTW. 32
5.7 File pairs affected by class- or package-level smells also reported as co-changing by FO. 33
5.8 File pairs affected by class- or package-level smells also reported as co-changing by DTW. 33
5.9 Percentages of projects for which at least one relation between smells and co-changes was found. 41
5.10 FO Results for projects with a large amount of overlapping file pairs. 41
5.11 FO Results for projects with a smaller amount of overlapping file pairs. 42
5.12 DTW Results for projects with a large amount of overlapping file pairs. 42

5.13 DTW Results for projects with a smaller amount of overlapping file pairs. 42

List of Tables

3.1 Analyzed projects for the case study. 12

4.1 Example contingency table. 24

5.1 Percentage of all file pairs reported as co-changing. Values over 5% are marked in bold. 27
5.2 Results of testing H^{RQ3} with co-changes reported by FO and class-level AS. The threshold values for the four conditions can be found in Section 5.3. 34
5.3 Results of testing H^{RQ3} with co-changes reported by DTW and class-level AS. The threshold values for the four conditions can be found in Section 5.3. 36
5.4 Results of testing H^{RQ3} with co-changes reported by FO and package-level AS. The threshold values for the four conditions can be found in Section 5.3. 37
5.5 Results of testing H^{RQ3} with co-changes reported by DTW and package-level AS. The threshold values for the four conditions can be found in Section 5.3. 38
5.6 Results of testing H^{RQ3} with co-changes reported by DTW and all AS. The threshold values for the four conditions can be found in Section 5.3. 39
5.7 Results of testing H^{RQ3} with co-changes reported by DTW and all AS. The threshold values for the four conditions can be found in Section 5.3. 40

B.1 Result of the hyperparameter analysis for all projects. 55

List of Abbreviations

Abbreviation	Meaning	Definition
AS	**A**rchitectural **S**mell	Section 1.1
CPI	**C**ode **P**roblem **I**ndication	Section 1.1
DTW	**D**ynamic **T**ime **W**arping	Section 2.1
FO	**F**uzzy **O**verlap	Subsection 3.4.1
MBA	**M**arket **B**asket **A**nalysis	Section 2.1
OP	**O**verlapping **P**airs	Subsection 3.5.4
TD	**T**echnical **D**ebt	Section 1.1
VCS	**V**ersion **C**ontrol **S**ystem	Subsection 1.2.1

Chapter 1

Introduction

1.1 Technical Debt

Technical debt (TD) was first introduced by Ward Cunningham in 1992 [12], defined as *"software development compromises in maintainability to increase short term productivity"*. It can be compared to regular debt, but instead of borrowing money, it is development time. Just like regular debt, technical debt often comes with interest. Even though a dirty yet quick implementation might save an hour right now, extending it down the road could cost another two hours instead of merely one.

Cunningham's definition does not state what these compromises might look like. Which implementations increase technical debt? And which implementations decrease technical debt? These questions have been subject to research themselves.

To help answer these questions, researchers have started to adopt metrics as indicators of the quality of a codebase and thereby of the amount of TD contained in it [22]. Two of these metrics in particular have been widely adopted [41, 27] and extended: coupling and cohesion. *Coupling* can be defined as the manner and degree of interdependence between software modules and *cohesion* as the manner and degree to which the tasks performed by a single software module are related to one another [23].

Coupling and cohesion can be measured in terms of modularity and dependencies. Low levels of modularity imply low cohesion, high levels imply high cohesion. The same relation exists between the levels of dependency and the amount of coupling. Low cohesion and high coupling are useful indicators for technical debt. To avoid these, concrete structures have been defined that yield better maintainability when they are correctly applied in a codebase [18, 14]: design patterns. These patterns can be seen as best-practice solutions for common problems.

For the aforementioned common problems, many other solutions exist. Some of these may be tempting to implement, yet they worsen maintainability. These are called *anti-patterns*, which have been documented as well as design patterns [43, 10]. Besides anti-patterns, there are so-called code smells: a code smell is a surface indication that usually corresponds to a deeper problem with the system [16]. Both elements are interesting when it comes to identifying technical debt.

A relatively new type of smell that has emerged in the field of TD is the architecture smell. *Architecture smells* (AS) are smells that indicate a problem within the

system's architecture [29], and they are usually caused by decisions that may not always have been intentional. Contrary to code smells, AS impact the maintainability of the system in the long term. Some examples of AS are: *Cyclic Dependency (CD)*, which occurs when two architectural modules depend on each other; *Hub-Like Dependency (HLD)*, which occurs when a component has many dependencies; and *Cyclic Hierarchy (CH)*: a supertype directly referring to a child type [6].

Although all anti-patterns, code smells, and AS indicate problems within a codebase, they are not the same. As stated above, anti-patterns are ineffective solutions to common problems. AS, on the other hand, are solutions to problems that are not properly implemented and thus negatively impact system quality. They consist of hidden structures in a system which only become visible through thorough analysis of the system's internal interactions. Both anti-patterns and AS differ from code smells in the sense that they appear on a higher level: they generally affect multiple components, whereas code smells can be found on the level of single lines of code. *Code Problem Indications (CPI)* will be used as an umbrella term for all three in this document.

1.2 Software systems evolution

1.2.1 Version Control Systems

In recent decades, researchers have started to look for more obscured indicators of TD. With the introduction of version control systems, new analysis types could be performed on systems to locate TD. Version control systems (VCS) track each change a developer makes to the system and, as a result, can recreate a consistent snapshot of any point in time [7]. Another term for such systems is Concurrent Version System (CVS), which is sometimes used instead of VCS [17]. This thesis treats them as the same and only uses the term VCS.

VCS are useful when working on systems with multiple collaborators/contributors residing at different physical locations. Typically, a system contains one *remote repository*, which is a centralized location where the source code and its revisions are stored. Collaborators *check out* (make copies/snapshots from) this remote state, thereby creating a *local repository* in which they can start making local changes. Changes that are made locally are tracked by the VCS. Once the collaborator wants to share their changes with other collaborators, the VCS bundles the tracked changes into a *changeset* (*commit*; *modification record*), which in turn can be checked in to the remote repository.

So what new information does a VCS offer? The most interesting added dimension is a project's *evolution over time*. A VCS facilitates the analysis of different snapshots of the same system at different points in time. A VCS also allows for zooming in on the difference between, for example, two major versions. Instead of just having two codebases, now a chain of time-labeled changesets which steps a state goes through in order to pass over into the next.

1.2.2 Co-changes

Researchers have used this new evolution information in several ways. One research topic that has emerged is the concept of co-changes. Co-changes find their origin in the term *logical coupling*, which is defined by Gall et al. as "observed identical change behavior of different elements during system evolution" [17].

Co-changes are noteworthy because they uncover *hidden dependencies* between artefacts. Two classes A and B may have 0 references to one another, but it might turn out that whenever class A changes, class B changes as well. With the introduction of VCS, co-changes could be mined on a lower level: that of changesets instead of versions. This is what makes co-changes stand out, seeing as changesets normally contain directly related changes.

1.3 Goals & Motivation

On average, the maintenance of a software system takes up more than 50% of its lifecycle time [42] and this maintenance comes down to, among other tasks, resolving technical debt. Curtis et al. estimate that a typical large scale application contains $3.61 of technical debt per line [13]. Since technical debt comes with interest, the detection and early resolution of technical debt can save money.

Previous research has already established a relation between co-changes and technical debt, as co-changes were found to be a good indicator of unexpected defects [40] and a complex change history in general was associated with a greater amount of faults in the system [20].

Although co-changes are technically a form of technical debt, they are merely a symptom. To find the underlying cause of co-changes, researchers have explored several potential causes of and relations between co-changes. They looked at code smells [33, 25], anti-patterns [26], and the relation between system faults and co-changes [40].

Since it is still difficult to accurately predict which files will co-change, exploring new relations might yield more predictive power. An interesting relation that still has unexplored territory is the one between AS and co-changes. Therefore, the first goal of this thesis is to bridge this gap and *explore the relation between AS and co-changes*.

As explained in Subsection 1.2.2, different approaches are being used to determine which file pairs are co-changing. No studies were found where more than one algorithm was applied to mine co-changes, so there exists little knowledge regarding the differences in reported co-changes by the algorithms being used. Therefore, the second goal of this thesis is to *explore the differences between co-change mining algorithms*.

1.4 Approach

To accomplish these goals, several applications and analyses will be used. Three different co-changes mining algorithms are implemented and applied on several open source projects: two existing algorithms (Market basket analysis (MBA) and Dynamic time warping (DTW)) and a new algorithm, *Fuzzy Overlap* (FO). A Java tool (`CoCo`) will be developed for this research. Furthermore, a pipeline will be set up to mine AS from the relevant open source projects, using existing tools. Finally, an analysis is carried out that analyzes the relationship between different co-change mining approaches and the relation between AS and co-changes.

1.5 Outline

The rest of this document is structured as follows: in Chapter 2, related work is discussed including studies related to mining VCS data, co-changes and different types of CPI. The next chapter (Chapter 3) entails the research questions, the design of the case study and an elaborate explanation of the *Fuzzy Overlap* algorithm. This chapter also contains a thorough analysis of the data collection. Chapter 4 then contains an in-depth description of how the analysis of the research questions will be executed. What follows (Chapter 5) are the results of the proposed research questions and finally a discussion (Chapter 6) and conclusion (Chapter 8).

Chapter 2

Related Work

In this section, previous research related to this thesis is discussed. First, a brief overview of the background of several co-change detection algorithms is given. Then, some uses of VCS data are discussed. This is followed by various applications of co-change mining. Finally, architectural smells are discussed, focusing on the most relevant research.

2.1 Background

An algorithm that is often used to determine association between changes of file pairs is Market-Basket Analysis (MBA) [47, 32]. As the name suggests, MBA analyzes a collection of baskets. A *basket* in the context of mining co-changes typically comes down to a changeset, which is a set of files being modified simultaneously. For all possible pairs of files in a codebase, two properties are calculated.

$$support = \frac{A+B}{total} \tag{2.1}$$

$$confidence = \frac{A+B}{A} \tag{2.2}$$

Support (Equation 2.1) measures how often (percentually) two files occur together in all (analyzed) changesets. High support indicates that the two files often change simultaneously with respect to all changesets. Confidence (Equation 2.2) measures how often file B changes whenever file A changes. High confidence indicates a possible directional relationship between both files. The apriori algorithm is commonly used to discover which pairs meet given thresholds for support and confidence. Apriori is a breadth-first algorithm that depends on the fact that a file set can never have a support value above threshold-level if any of its subsets does not reach the threshold. This allows the algorithm to quickly prune the list of candidates, leading to an efficiently calculated set of changesets that meet the threshold.

Other researchers use Dynamic Time Warping (DTW) for mining co-changes. DTW is more lenient than MBA in acknowledging co-changes over multiple changesets. This leniency stems from the fact that it compares the change histories of files as a whole and does not require exact overlap in changesets.

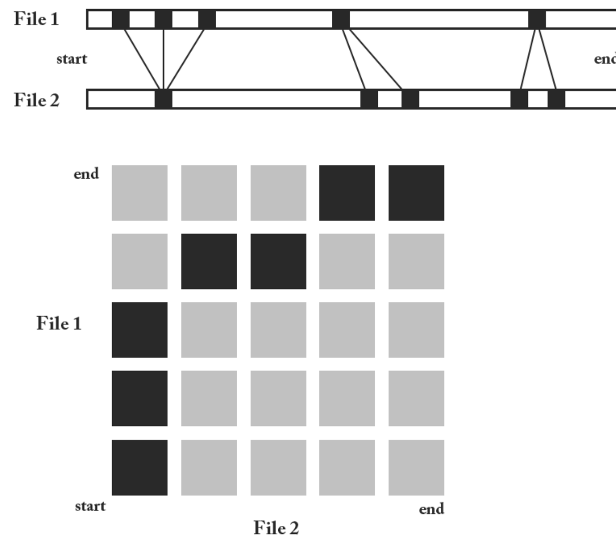

FIGURE 2.1: Mapping between changeset histories of two files [45].

Figure 2.1 shows the inner workings of the DTW algorithm. Assume a system contains two separate files: file 1 and file 2. Through the history of their system, both files were modified in certain changesets. This is represented as the two timelines at the top of Figure 2.1. The black dots represent the points in time (i.e. changesets) in which the files were changed. The bottom half of Figure 2.1 shows the warp matrix.

The algorithm works as follows: the warp matrix contains all distances (measured in time) between the changesets pertaining to file 1 and file 2. Starting at the left bottom square, the three adjacent squares (top, right, top-right) are considered as the next steps in the warp path. The one with the lowest value is included. This process is repeated until the end of both paths (the top right corner of the matrix) is reached. The result is a path in which changesets are mapped to the closest (in time) changesets of the other file. For DTW the difference in change history between the two files is the length of the path, often normalized against the length of the change history. This value can be held against a threshold and, if low enough, the file pair is marked as co-changing.

Note that DTW could very well mark the example of Figure 2.1 as co-changing, whereas MBA most likely would not; the path chosen by DTW shows that the changes of both files rarely overlap, even though they are close in time. The total difference would still be small, even with just one overlapping changeset between the files. MBA is bound only to exact matches in changesets and would rate this pair less likely to co-change.

2.2 VCS

VCS offer datasets rich in information that can be used for all sorts of research. Various studies have therefore been conducted examining the evolution of systems over time. For example, Zimmermann et al. created a tool that extracts coupling between artefacts and is able to predict future changes [47]. Mockus et al. started

to look at changesets more in-depth [31]. They developed a system that labeled changesets according to the type of change it made. Weißgerber et al. shed light on a different part of data offered by VCS [44]. They found that, for some projects, certain developers only worked on certain modules of the system and were able to mark one or more contributors as the main developers of a project.

2.3 Co-changes

Researchers tend to agree that co-changes describe similarly changing artefacts. However, this definition lacks clear boundaries as to what 'similarly' exactly means. Over the years, researchers have therefore used a variety of approaches and boundaries in mining co-changes.

Jaafar et al. propose two types of co-changes: MCC (Macro Co-Changing) and DMCC (Dephase Macro Co-Changing) [24]. These concepts describe two files changing simultaneously (MCC) or nearly simultaneously (DMCC). The authors explain what Macocha, their approach to mining these co-changes, does. They attempt to find files that are MCC or DMCC using a sliding window approach, 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 manage 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 changesets, resulting in a typical horizon effect. 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. [47] also look at mining co-changes using Market Basket Analysis (MBA). Every changeset is treated as a 'basket' containing several changes. Using the apriori, algorithm they are able to mine association rules from histories of these changesets. 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.
Ajienka et al. mine co-changes from VCS and relate those to semantic coupling [1]. They also use MBA to mine the co-changes and have found that certain semantic relations between artefacts are related to artefacts being co-changing.
Mondal et al. use MBA to mine co-changing method groups [32]. They analyze the changesets of 7 open source projects and compare the lifetime and change-proneness of co-changing methods with those of not co-changing methods. They have 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 [35]. 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, the research for this thesis utilizes 'traditional' VCS data.

2.4 Architectural Smells

(Architectural) smells find their origin in the code smells and anti-patterns. Anti-patterns date back to the 1990s when Webster [43] and Brown et al. [10] published books which laid out guidelines for development in an object-oriented fashion and provided a catalog of code structures to avoid or remove during development. Code smells were first introduced in the early 2000s as part of a manual on refactoring [15].

Other researchers investigated relations between CPI and other codebase-attributes. Closely related to the goal of this thesis, Khomh et al. explore the relation between code smells and software change-proneness [25] as well as the relation between anti-patterns and software change-proneness [26]. They find that smelly artefacts (artefacts containing a code smell) are more prone to change. Similarly, they conclude that artefacts containing anti-patterns are in general more prone to fault and change than other artefacts.

While many studies focus on class- and package-level CPI, some research focuses on other levels. Brown et al. already labeled their anti-patterns with seven different levels on which they can occur [10]. CPI can, for example, only affect either a single class (object level), multiple classes (micro-architecture/package level) or entire systems (system level). Fontana et al. researched so-called micro patterns, which entail patterns as small as single method invocations [3]. They have developed a plugin for Eclipse that detects these patterns in codebases and connect the found anti-patterns to code quality metrics of the respective codebases. Hecht et al. focus on class-level smells. Their tool Paprika automatically detects eight anti-patterns in Android apps [21]. Nayrolles et al. then take on a higher-level perspective [34], by analyzing Service Based Systems (SBSs) in order to locate SOA (Service Oriented Architecture) anti-patterns. These patterns are mostly based on incoming and outgoing coupling.

Garcia et al. introduce the term *architectural bad smell* and propose four instances of such AS [19]. In continuation, other catalogs of such AS are proposed [29, 28] and combined [6].

Subsequently, tools were developed for detecting AS in codebases [30, 5]. One of these, Arcan, requires special attention. Created by, Fontana et al., this tool is able to detect three types of AS in codebases [5, 4]: Cyclic Dependencies, Hub-Like Dependency and Unstable Dependency. The tool analyzes a system's source code per version and outputs its results as a graph of smells and affected artefacts.

While Arcan's output has by itself been used for predicting future AS [2], it can also be used by a second tool: ASTracker [37]. ASTracker shows the evolution of smells

throughout the history of a system. In the paper in which the tool was introduced [37], several attributes of AS were discussed. With regards to some of their findings, they stated that AS tends to move to the more centrally located parts of a system over time, while generally staying the same size.

Sas et al. intend to use the output of ASTracker [38]. Their research (at the moment of writing still under review) investigates relations between change rate and change size of artefacts by comparing them against the evolution of the architectural smells in which they are or are not contained. This thesis is closely related to this research, as it explores how co-changes relate to architectural smells and their evolution.

Chapter 3

Research Methodology

This chapter explains the design of the case study used for this research. Then the goal and the research questions of this study are discussed. Finally, the methods used to collect all the data necessary for the analysis are described.

3.1 Case Study Design

This case study is set up according to the guidelines for case study design as described by Runeson et al. [36]. The general structure is displayed in Figure 3.1. In this study, software projects function as cases and their packages and files function as units of analysis. By analyzing 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. Software projects can vary in size, style and structure and this is why the results of a case study with a single case are not significant, as these results can hardly be projected onto other projects. Analyzing a single unit of analysis per case yields similar problems, seeing as different units can internally also be radically different from others in many aspects. Therefore, we deem it important to review many different units per case, and so end up with the embedded multiple-case study setup.

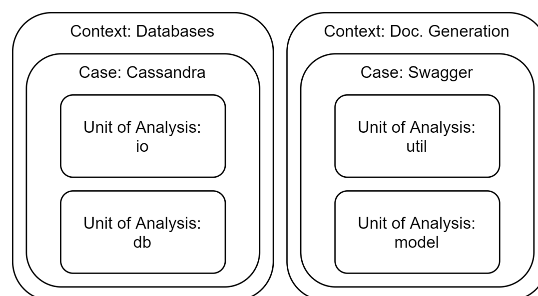

FIGURE 3.1: Setup of the case study using examples.

3.2 Goal and Research Questions

The goal of this study is to understand the relation between co-changes and architectural smells. In order to do so, architectural smells have to be extracted and co-changes need to be identified. This study serves as a continuation of a previous study done by Sas et al. [38] which locates architectural smells according to specific metrics. This process is performed by an application called *ASTracker*.

Our goal is to analyze the change history of projects by means of identifying co-changing artefacts, comparing different co-change detection algorithms, and by ultimately studying the relation between co-changes and architectural smells. To achieve this goal, we have formulated four research questions:

- **RQ1** *What are the differences between the three co-change detection algorithms FO, MBA, DTW?*
- **RQ2** *Are artefacts affected by architectural smells co-changing?*
- **RQ3** *Are co-changes more often found in smelly artefacts?*
- **RQ4** *Are smells introduced before or after files start co-changing?*

RQ1 requires us to compare the output of the FO algorithm for detecting co-changes with the output of MBA and DTW. Both MBA and DTW have their own recommended values for their thresholds based on earlier research. The results of this comparison will offer insight into the accuracy of the novel FO algorithm and its impacts on the results of the other research questions compared to state-of-the-art algorithms.

RQ2 requires us to explore co-change and smell data and the overlap between them. We will investigate which artefacts are affected by smells and which of those also co-change. The results of this data exploration will likely generate insights that are useful for answering the other RQs and form a critical piece of the puzzle in understanding the relation between AS and CC.

RQ3 can be considered an extension of RQ2. Here, we not only look at the co-occurrence of smells and co-changes, but also attempt to determine whether a relationship between them exist.

RQ4 requires us to compare the starting dates of co-changes and code smells. Based on those results, we check which one of the two typically occurs before the other. This will give us insight into the temporal emergence of co-changing file pairs that are also part of architectural smells.

3.3 Analyzed Projects

As described in the introduction of this section, software projects can differ from each other considerably. Analyzing a wide variety of projects (cases) for our study is therefore important. Runeson et al. present different methods for selecting projects [36]. For our study, we have adopted the method of aiming for *maximal variance*

between our cases. We want to achieve variance in the distribution of the following properties:

- *Project size* - We want to analyze projects with a smaller amount of artefacts and projects with a larger amount of artefacts.
- *Domain* - We want to analyze projects within different domains.
- *Owner* - We want to analyze projects belonging to different owners and authors.

Project	Description	Owner	Changesets	Domain
ArgoUML	UML modelling tool	Tigris-org	17814	Documentation
Cassandra	NoSQL rowbased database	Apache	25081	Databases
Druid	Realtime analytics database	Apache	10202	Databases
Hibernate-ORM	Object Relational Mapping	Hibernate	10167	Databases
Jackson-databind	JSON library	FasterXML	6387	Formatted Data
JUnit5	Unit testing framework	JUnit-Team	6143	Testing
MyBatis-3	SQL object mapper	MyBatis	3346	Databases
PDFBox	PDF manipulation	Apache	8778	Formatted Data
POI	MS Office interaction	Apache	10163	Formatted Data
PgJDBC	Postgresql Java Driver	Pgjdbc	2638	Databases
Robolectric	Android unit testing	Robolectric	10101	Testing
RxJava	Reactive JVM Extensions	ReactiveX	5741	General purpose
Sonarlint-IntelliJ	Linter for IntelliJ	SonarSource	1102	General purpose
Spring-Framework	Enterprise framework	Spring-Projects	20479	General purpose
Swagger-Core	API-documentation	Swagger-API	3683	Documentation
TestNG	Testing Framework	Cbeust	4752	Testing
Xerces2-j	Java XML parser	Apache	6374	Formatted Data

TABLE 3.1: Analyzed projects for the case study.

3.4 Approaches to detect co-changes

In order to validate and compare the results of our algorithm, we compare our output with the output of two other commonly used algorithms. These are the Market Basket Analysis (MBA), using the apriori algorithm, and a Dynamic Time Warping (DTW) algorithm, which calculates the distance between two commit sets. From this moment on, *commit* will be used rather than *changeset*, seeing as all analyses are performed on data gathered using Git, which uses “commit” to refer to a changeset.

3.4.1 Fuzzy Overlap

The Fuzzy Overlap (FO) algorithm is an algorithm that tries to formalize certain intuitive assumptions regarding co-changes in a domain-focused fashion. These assumptions cannot be satisfied using more generic algorithms such as MBA and DTW.

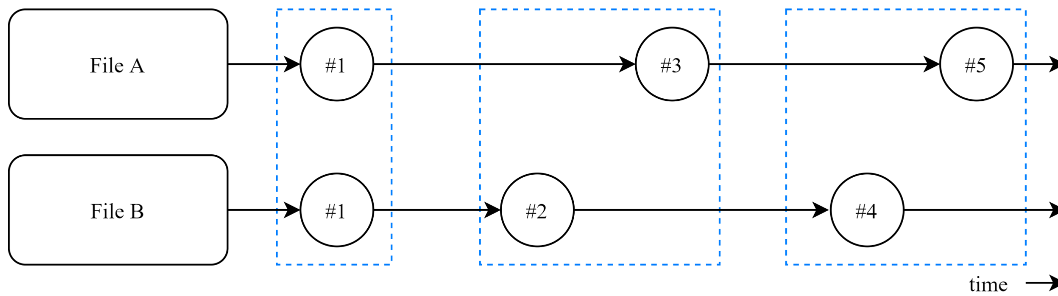

FIGURE 3.2: The basic idea of fuzzy mapping. The circles represent commits in which the files changed. In commit #1, both files change. After that, they do not change in the same commits, but file B always changes just before file A.

This algorithm 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, then a relationship between the two might exist and intuitively these two files would then be considered to be co-changing. This is depicted in Figure 3.2. The MBA algorithm cannot pick up on these kinds of co-changes as it only incorporates changes within the same commits.

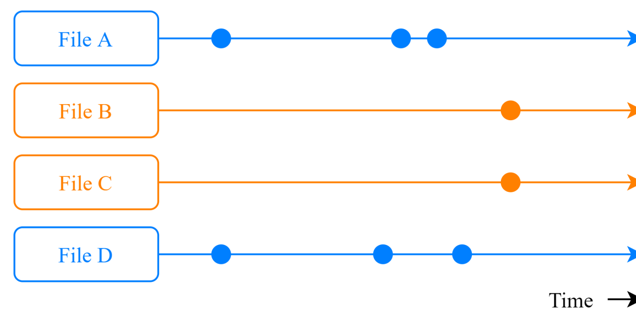

FIGURE 3.3: This figure compares two typical file history types. Files B and C (orange) are spotted by DTW as co-changing as their history is identical. However, the *support* of the similarity is minimal. Files A and D are less likely to be matched by DTW as their paths differ; however, FO will pick up such file pairs.

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. An example of this can be found in Figure 3.3. MBA can filter out these changes due to the support threshold.

This algorithm is implemented in the form of a Java application called `CoCo`[1]. `CoCo` acquires co-changes in two steps. First, it collects the commit history of a project, optionally starting from a certain date, and up to a certain commit. This is done in order to limit projects that have a great amount of commits compared to other projects analyzed. For all files present at a certain point in the project, `CoCo` determines by which commits they were affected (added, modified or moved to a different directory). The raw change data is the collection of all changes within the given time window, together with their dates. Commits related to the merging of two Git branches are excluded, as these should be considered noise, based on research by Zimmerman et al. [46].

Next, the accepted co-changes are generated from the raw changes. For every pair of files, the changes are compared in order to spot overlapping commits. An instance of the files changing in the same commit is always considered a co-change, but files that change shortly after one another can also be co-changing. In order to limit these changes over time, two variables must be considered. The first is the time difference between two commits. This parameter needs to be flexible, as different projects have different times between commits. The second is the number of commits in between the two commits. If there is a large number of commits in between them, this instance is less likely to be a co-change as compared to two changes that occur immediately after each other.

Finally, the co-changes are compared to a final filter, that of the co-change threshold. The goal of this threshold is to filter out noise and co-changes occurring purely by chance.

Pseudocode for the FO algorithm can be found in Listing 3.1 and a visual overview can be seen in Figure 3.4.

[1] `https://github.com/RonaldKruizinga/CoSmellingChanges`

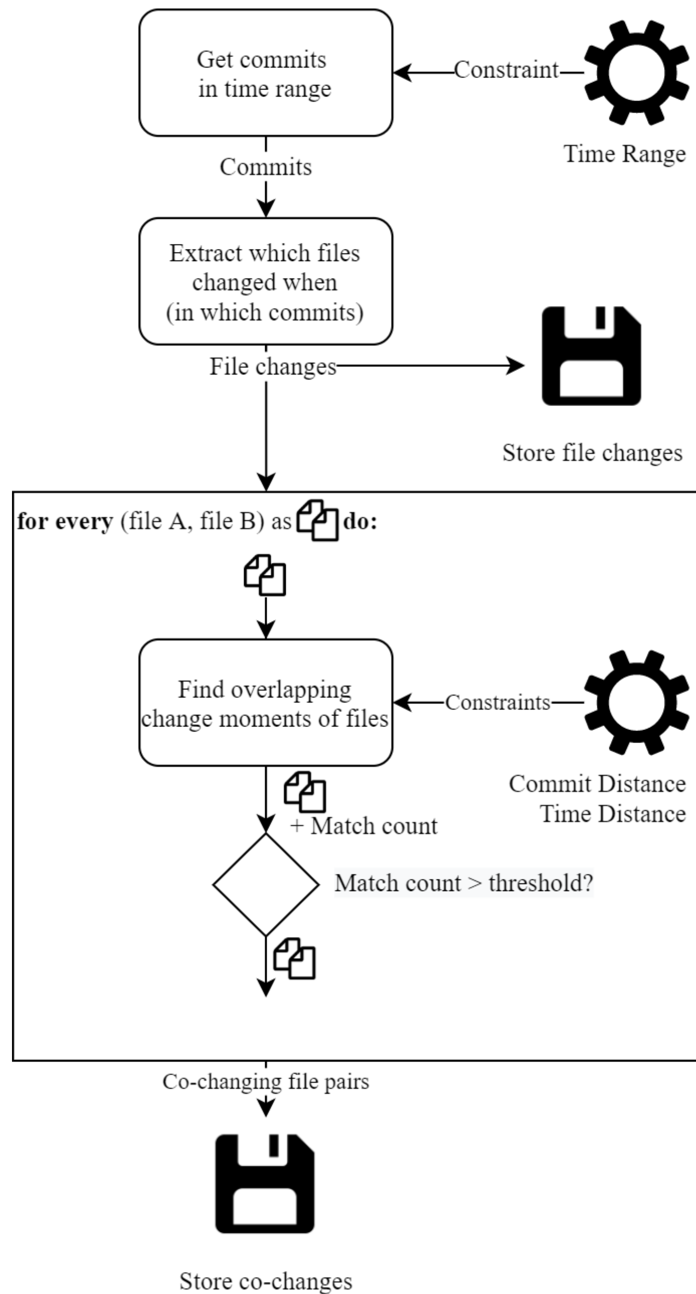

FIGURE 3.4: Flowchart of the FO algorithm.

Hyperparameters

From Subsection 3.4.1 several hyperparameters can be selected which require exploration to find the optimal values. As soon as we determine appropriate values for each of them, we can start analyzing the space bounded by these values. The following hyperparameters are important for the FO algorithm:

- **Commit Distance** - *The number of commits between two analyzed commits.*
- **Time Distance** - *The maximum time between two commits for them to be marked as co-changing.*

- **Time Range** - *Interval of development time which bounds the sampled commits.*
- **Match Threshold** - *The minimum number of overlapping commits of two files for them to marked as co-changing*

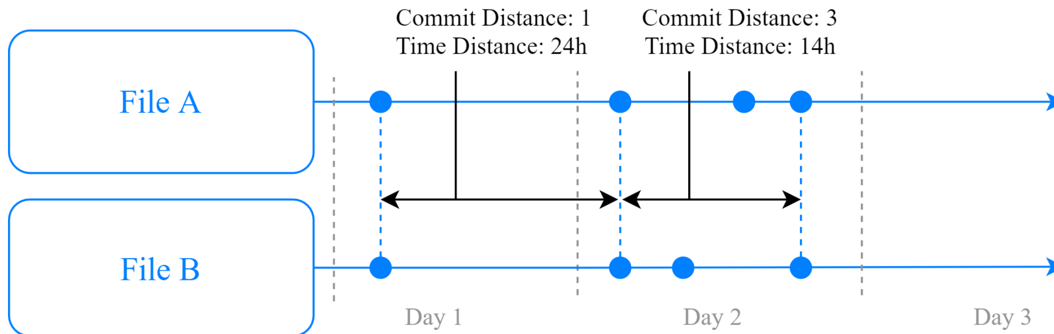

FIGURE 3.5: Difference between commit distance and time distance hyperparameters used by FO.

Commit Distance

This hyperparameter is one of the two constraints that bounds the leniency of the co-change detection. Setting it to a higher value means that more files will be marked as co-changing, since more distance may be present between two commits to be marked as related. Lowering the value tightens the detection. Ultimately, when this number set to 0, files must change within exactly the same commit to be marked as a co-change. This parameter is set by looking at how many commits there are on the average, on a day which has commits, as that gives an indication of the grouping of the commits. This average therefore ignores days without commits.

Time Distance

While this hyperparameter also constrains the distance between two commits, it calculates a commit's distance in time rather than counting the number of intermediary commits. Note that the same configuration of these two distance metrics can yield notably different results in different projects. Some projects contain relatively few commits in their main branch, each with a large amount of time in between them. Others might not stick to a few strict releases and accept pull requests on a daily basis. A larger commit distance does not necessarily equate to a large timespan between commits, as some projects have five commits on a single day, whereas others have five spread out over a week. With regards to the time distance, the same is true the other way around: a time distance of a week might be necessary to capture all co-changes in the first type of projects, while in the second type of project this might result in a far too lenient detection procedure. This parameter is set by looking at the time intervals between commits and taking the third quartile of those, as suggested by Bird et al.[11].

Time Range

This hyperparameter determines the overall time range that is analyzed in the VCS, based on a start date and an end date. If this parameter is too broad, especially with high distance bounds, it might yield irrelevant co-changes. If the window is made too small, it is possible that essential co-changes are left out. The optimal value lies somewhere in between and highly depends on the project analyzed.

Match Threshold

Given a large enough time window, most files will co-change occasionally. We are not interested in coincidental co-changes, but rather in the ones that occur repeatedly. The goal is to find the co-changes that occur more often than merely at random and we want to only keep these co-changes in our results by filtering out ”noise” (=coincidental co-changes).

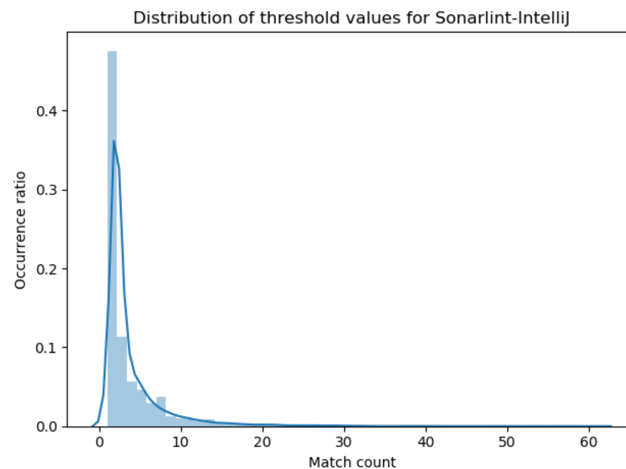

FIGURE 3.6: Distribution of observed match counts among co-change candidate pairs. A threshold of 12 was selected for this dataset.

We therefore introduce a threshold which maintains a required minimum amount of overlapping commits between two files needed to be marked as co-change. When set correctly, this hyperparameter filters out noise; however, if it is set too high or too low, it yields too few or too many results. This hyperparameter is found by first running `CoCo` without a threshold. This yields co-change candidates: file pairs with at least one (fuzzy) overlapping commit. The distribution of match counts occurring in this dataset is typically exponentially distributed, as shown in Figure 3.6. The match count at the 95th percentile of this dataset is chosen as the threshold for the project. This is based on related research [11] and on empirical experiments.

```
Calculate_FO_CoChanges(githubRepository){
  for each commit in githubRepository within time range{
    Calculate fileChanges and store them
  }

  Export changes to CSV to be used by DTW and MBA

  // i is a file with its changes
  for i in fileChanges{
    // j is a file with its changes that comes after i in the list
    for j > i in fileChanges{
      // All combinations consist of one commit from i and from j
      Calculate all combinations of two commits
      // No more than x hours apart
      Filter commit pairs on time distance
      // No more than x other commits inbetween
      Filter commit pairs on commit distance
      if number of incidents exceeds threshold{
        Store co-change
      }
    }
  }

  for each co-change{
    Calculate start and end date
  }

  Export co-changes to CSV to analyze
}
```

LISTING 3.1: FO pseudocode

3.4.2 Market Basket Analysis

The market basket algorithm is based on association rules and calculates how often two items in a set occur together and whether the presence of one item is likely to predict the presence of another item in the same basket. This algorithm has traditionally been used to analyze which items are commonly bought together and what kind of promotions would therefore be likely to improve sales.
This is calculated using the apriori algorithm, limited on sets of length two (one antecedent, one consequent) in order to improve performance. Based on a 2013 case study by Bavota et al. [8], a good value for the support for a rule is at least 2% of the commits, with a confidence of at least 80%. These values are chosen to pick up changes in a low amount of commits (hence the low support), yet with a high confidence to still ensure preciseness. For the purpose of comparing algorithms, these values will therefore be used.

3.4.3 Dynamic Time Warping

Another alternative algorithm is that of dynamic time warping, which 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 normalized version of the distance. This is the value we use to threshold and find co-changes.
Based on a 2006 case study by Bouktif et al. [9], a threshold of 86400 seconds provides a good balance between accuracy and performance.

3.5 Data Collection

Various sorts of information need to be collected for the analysis, such as information on the architectural smells and the co-changes within a project. The calculations will be run on the Peregrine cluster of the University of Groningen.

3.5.1 Architectural smells

For this research we will relate co-changes to architectural smells. The architectural smell data will be collected using the tools ASTracker[2] and Arcan[3]. ASTracker analyzes projects per *version*. The term "version", in this context, refers to snapshots of the source control taken at given time intervals. We set this interval to the minimum value, being one day, which means ASTracker will analyze every day a project was modified.

The collected smell data tells us which artefacts are affected by a smell (this can be one or several artefacts per smell) and tells us during which commit(s) the smell was present. This procures a time range during which a smell was present. It also stores the date of each version.

Smell-scopes

Every smell has its own *level*: class or package. The reported affected artefacts are of one of these types. We distinguish three *smell-scopes* in this context: class (C), package (P), and class plus package (CP). The scopes correspond with a subset of the collected smell data where one or both levels are included.

Smell-affected pairs

As our research focuses on the relationship between *smelly (file) pairs* and co-changing file pairs, we need to convert the smells to file pairs that are affected by the smells. For class-level smells, this is merely a matter of creating all combinations of files affected by the smell, thereby creating a set of affected file pairs. For package-level smells, however, this process is somewhat more complicated; since entire packages are affected by the smell, all file combinations between those packages are considered affected pairs.

[2] `https://github.com/darius-sas/astracker`
[3] `https://gitlab.com/essere.lab.public/arcan`

3.5.2 File Changes

CoCo is capable of tracking when certain files change over time. It outputs this information per file and per version (in our case, per commit). This is used as input for the DTW and MBA algorithms that will be used during the analysis phase.

3.5.3 Co-changes

All three algorithms have a similar way of generating co-changes from the list of changes. For FO, this is done within CoCo. For MBA and DTW, this is done in the analysis script we have written in Python[4]. We analyze all systems listed in Table 3.1 and gather information regarding co-changes from them.

Fuzzy Overlap

Data collection is done using the Java CoCo application, wherein we collect raw co-change data and then apply a collection of filters (defined per project by the hyperparameters in Section 3.4.1) to obtain the accepted co-changes of each project.

The input for this algorithm are the hyperparameters discussed in Subsection 3.4.1 and metadata regarding the location of the Git repository. All co-change data for this algorithm is outputted as a CSV containing the list of co-changing file pairs, including the packages in which the files can be found. This file also contains a start and end date for each co-changing file pair. These dates are determined by the first date and the last date on which the file pair reports a co-change.

Market Basket Analysis

Data collection is performed using the mlxtend[5] package for Python. This package implements all the necessary algorithms including flexible support and confidence thresholds. The values for these thresholds are discussed in Subsection 3.4.2.

The input for this algorithm is the list of changes outputted by CoCo. The algorithm's output is then stored as a CSV containing the list of co-changing file pairs, including the packages in which the files are found. This file also contains a start and end date for each co-changing file pair. These dates are determined by the first date and the last date on which either of the files in the pair reports a change.

Dynamic Time Warping

Data collection is performed using the dtw-python[6] package, which provides the necessary time warp comparison, including a normalized distance threshold and plotting functions to visualize the results. The threshold value is discussed in Subsection 3.4.3.

[4]https://github.com/rubenscheedler/CoChangeAnalysis

[5]http://rasbt.github.io/mlxtend/api_subpackages/mlxtend.frequent_patterns/

[6]https://dynamictimewarping.github.io/python/

The input for this algorithm is the list of changes outputted by CoCo. The output of the algorithm is then stored as a CSV containing the list of co-changing file pairs, including the packages in which the files are found. This file also contains a start and end date for each co-changing file pair. These dates are determined by the first date and the last date on which either of the files in the pair reports a change.

3.5.4 Overlapping pairs

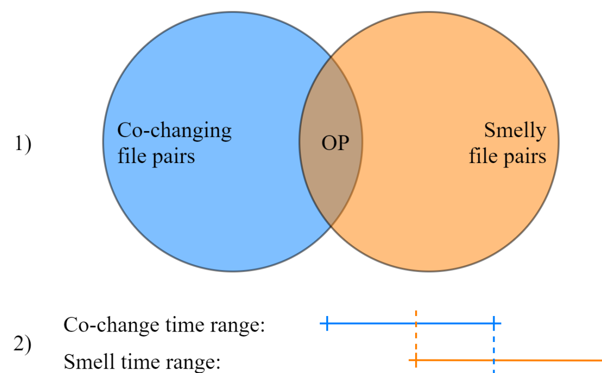

FIGURE 3.7: The two conditions for an overlapping pair (OP).

The final piece of data we need to collect is the overlap between file pairs that are co-changing and file pairs that are part of an architectural smell. We call such pairs *Overlapping Pairs* (OP). The process of collecting these pairs is the same for all three algorithms. First, the list of co-changes is matched with the list of file pairs affected by architectural smells on filenames. These smells can be on either class or package-level, or both, as discussed in Subsection 3.5.1. Secondly, the time ranges of a pair's co-change and smell are checked for overlap. A co-change's time range starts at the first (fuzzy) overlapping change and ends at the last. A smell's time range starts at the first commit in which it is detected and ends at the last in which it is detected. The two conditions for an OP are visualized in Figure 3.7.

Chapter 4

Data Analysis

As mentioned before, our goals are to compare co-change mining algorithms and to determine whether a relationship between co-changes and architectural smells (AS) exists. These goals are covered by our four research questions. First, a data exploration into smells and co-changes and the various algorithms that are necessary to detect those co-changes is performed. Next, we perform a statistical analysis in an attempt to discover a correlation between co-changes and AS. Finally, we perform an analysis to determine whether AS do, in fact, occur before co-changes rather than after. The scripts used for these analyses can be found on GitHub [1].

4.1 RQ1 - What are the differences between the three co-change detection algorithms FO, MBA, DTW?

In order to answer **RQ1**, we need to perform an exploration and comparison of the three co-change detection algorithms. This analysis concerns the set of co-changes outputted by each algorithm. Moreover, each project is analyzed separately. We then compare the overlap between the algorithms and which percentage of all file pairs is reported as co-changing.

Every co-change is bound to a time interval, which is reported together with the files it concerns. Different algorithms might report different time intervals for the same files. Moreover, different algorithms might report more co-changes in certain periods of a project's history. For RQ1, we therefore also look at the distribution of co-changes over time.

4.2 RQ2 - Are artefacts affected by architectural smells co-changing?

For this research question we will construct two datasets. The first consists of the the file pairs affected by at least one architectural smell. The second is a subset of the first, created by determining which file pairs are also marked as co-changing. **RQ2** is answered by looking at the ratio between these two datasets and by calculating

[1] `https://github.com/rubenscheedler/CoChangeAnalysis`

which percentage of the architectural smells is also co-changing.

$$StartDate_{co-change} \leq EndDate_{smell} \leq EndDate_{co-change} \tag{4.1}$$

$$StartDate_{Smell} \leq EndDate_{co-change} \leq EndDate_{smell} \tag{4.2}$$

Every detected architectural smell and every co-change has a date-range during which it was observed. In order for a file pair to be considered a match, the relevant period of its smell and co-change has to overlap. A smell and co-change are considered overlapping once Equation 4.1 or Equation 4.2 holds.

RQ2 serves mostly as an exploratory question leading up to RQ3. It provides insight in where the most overlap is found between AS and co-changes. Therefore, this analysis is performed for all three algorithms (FO, MBA, DTW) and all three subsets of AS (class-level, package-level and both together). This will yield nine overlap ratios per project.

4.3 RQ3 - Are co-changes more often found in smelly artefacts?

To answer **RQ3**, we strive to compare two binary categorical variables. The first variable is whether a file pair is co-changing. The other 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 exist to determine this. The two best candidates are the χ^2-test and the Fisher's exact test.

Based on the size of our data set, we have decided to go with the χ^2-test. Fisher's test is best to be used with a sample size ≤ 20 [39]. Our data set is orders of magnitudes larger as our sample consists of all possible pairs of files in a repository (changed in the relevant time frame) meaning this test would be unsuitable. For our analyzed projects, the amount of pairs per project lies somewhere between 10,000 and 2,000,000.

The input for a 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 4.1.

The null hypothesis and alternative hypothesis are as follows for this research question:

- $H_0^{RQ3_[algorithm]_[scope]}$: Artefacts affected by AS are as likely to co-change as artefacts not affected by AS.
- $H_1^{RQ3_[algorithm]_[scope]}$: Artefacts affected by AS are more likely to co-change than artefacts not affected by AS.

RQ3 will be answered for all three co-change detection algorithms and all three smell-scopes. In total, RQ3 will thus be answered nine times, namely for every combination of algorithm (FO, DTW, MBA) and smell-scope (C, P, CP). These parameters are added as labels ($[algorithm]$ and $[scope]$) to the hypotheses when considering only one algorithm and/or smell-scope.

Normally, one would reject H_0^{RQ3} 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 Equation 4.3.

$$\phi = \sqrt{\frac{\chi^2}{n}} \tag{4.3}$$

In Equation 4.3, 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: $0.1 \leq \phi < 0.3$ means a small effect, $0.3 \leq \phi < 0.5$ means an average effect and $\phi \geq 0.5$ means a large effect [39]. To reject H_0^{RQ3} ϕ needs to be ≥ 0.1.

Moreover, to accept H_1^{RQ3}, we need to know the direction of the association our test might find. We calculate its odds ratio.

	Co-changed	**Not Co-changed**
No Smell	x	z
Smell	w	y

TABLE 4.1: Example contingency table.

Say we have Table 4.1 and want to calculate the odds ratio of this, we use the following formula:

$$o = \frac{x * y}{w * z} \tag{4.4}$$

o can vary between 0 and infinity. When it is larger than 1, it indicates a higher than 50 percent chance that a co-change will be present whenever a smell is present. Since H_1^{RQ3} states that containing a smell increases the change for a co-change, we can accept this when o (odds ratio) is larger than 1 [39].

4.4 RQ4 - Are smells introduced before or after files start co-changing?

In order to answer RQ3, it is required to find overlap in time between co-changing artefacts and artefacts affected by architectural smells. The file pairs which overlap can be investigated further. Most pairs are not permanently co-changing, nor smelly. RQ4 raises the question whether one of these tends to happen before the other, or whether they happen at the same time. For all overlapping pairs, it can be determined whether they first start smelling or first start co-changing. The dataset of overlapping co-changes and AS can be categorized into three groups:

1. $Emergence_{smell} < Emergence_{co-change}$ *(smell-earlier)*
2. $Emergence_{smell} > Emergence_{co-change}$ *(co-change-earlier)*
3. $Emergence_{smell} = Emergence_{co-change}$ *(neither-earlier)*

$Emergence_{smell}$ is the date of the commit in which the smell is introduced. $Emergence_{co-change}$ is the date of the first commit in which both files of the co-change changed. Not all file pairs are suitable for this analysis. For certain projects, only part of their history was analyzed. This means that co-changes and smells detected at the start of the analyzed window might have emerged earlier. Co-changes and smells for which this holds are therefore ignored. Furthermore, co-changes and smells that have no overlap are also left out of this analysis.

RQ4 indirectly asks for two things to be explored: are file pairs co-changing before they start smelling (1) or are file pairs smelly before they start co-changing? (2). These become two separate sub-RQs:

RQ4a - Are smells introduced before files start co-changing?
RQ4b - Are co-changes introduced after files start smelling?

The dataset for both RQ4a and RQ4b is binomially distributed since, for every file pair, one of two things holds: success, where one phenomenon indeed precedes the other, and failure, for which this is not true. Ties are logically included in the failure group. The binomial distribution implies that RQ4a and RQ4b can be answered using the binomial sign test [39].

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.

Say that π_1 is the probability of a pair falling in category 1, π_2 the probability of it falling into category 2 and $\pi_1 + \pi_2 = 1$. A null hypothesis can then be formed based on the expected value for π_1. This value is set to 0.5, 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 in the *skewing direction* if it does not match. Therefore, two one-tailed tests are used instead of one two-tailed test. This gives rise to the following hypotheses:

RQ4a - Are smells introduced before files start co-changing?

$H_0^{RQ4a_[algorithm]}$: $\pi_s \leq 0.5$

$H_1^{RQ4a_[algorithm]}$: $\pi_s > 0.5$

RQ4b - Are co-changes introduced after files start smelling?

$H_0^{RQ4b_[algorithm]}$: $\pi_c \leq 0.5$

$H_1^{RQ4b_[algorithm]}$: $\pi_c > 0.5$

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

For RQ4a and RQ4b, the null hypothesis is rejected when two conditions are met. Firstly, earlier smells and earlier co-changes must occur more often, respectively, for RQ4a and RQ4b. Secondly, the probability of the observed amount of successes *or more* must be lower than 0.05. Say, for example, for RQ4a that *m* smells occurred earlier and *n* co-changes. H_0^{RQ4a} 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^{RQ4a} or H_1^{RQ4b} can be accepted.

Chapter 5

Results

This chapter covers the results obtained in our research. They are reported according to their corresponding research question and the significance of each result is discussed. All results have been collected and analyzed with the methods described in Chapter 3.

For *Cassandra*, *Hibernate-ORM* and *Spring-Framework*, memory constraints made it impossible to calculate the Overlapping Pairs. Due to this, RQ2, RQ3 and RQ4 did not consider these projects. In addition, similar memory constraints made it impossible for RQ4 to be answered for *ArgoUML*, *PDFBox*, *POI* and *Robolectric*.

5.1 RQ1 - What are the differences between the three co-change detection algorithms FO, MBA, DTW?

Project	% of files (number)			**Total file pairs**
	FO	**DTW**	**MBA**	
ArgoUML	4.48 (140,710)	0.55 (17,258)	x (0)	3,140,960
Cassandra	0.80 (22,515)	0.14 (4,056)	< 0.01 (1)	2,811,467
Druid	3.73 (69,567)	2.05 (38,259)	x (0)	1,866,807
Hibernate-ORM	2.15 (137,411)	1.71 (108,771)	x (0)	6,378,904
Jackson-databind	2.46 (3,474)	0.3 (497)	x (0)	141,353
JUnit5	3.45 (11,506)	0.74 (2,477)	< 0.01 (1)	333,580
MyBatis-3	**38.22** (25,497)	0.19 (126)	x (0)	66,703
PDFBox	0.76 (2,790)	0.12 (470)	x (0)	368,982
PgJDBC	**12.25** (17,247)	0.17 (236)	< 0.01 (6)	140,824
POI	1.52 (11,404)	0.27 (2,029)	x (0)	747,846
Robolectric	2.14 (41,071)	0.06 (1,236)	x (0)	1,918,436
RxJava	3.24 (43,457)	*4.07* (54,644)	x (0)	1,341,238
Sonarlint-IntelliJ	3.58 (1,109)	0.26 (82)	< 0.01 (1)	30,987
Spring-Framework	**5.02** (380,201)	0.84 (63,456)	x (0)	7,566,671
Swagger-Core	2.35 (1,395)	1.13 (673)	0.01 (4)	59,439
TestNG	2.85 (69,047)	***8.03*** (194,655)	x (0)	2,425,206
Xerces2-j	3.37 (8,792)	2.19 (5,716)	x (0)	260,670

TABLE 5.1: Percentage of all file pairs reported as co-changing. Values over 5% are marked in bold.

All three algorithms were run on the file changes reported by `CoCo` and report a large variety of co-changes, as can be seen in Table 5.1. With the exception of four results regarding the *MyBatis-3*, *PgJDBC*, *Spring-Framework* and *TestNG* projects, all results were below 5% of all pairs.

Considering the various algorithms, MBA reported a maximum of 6 co-changes for the *PgJDBC* project and did not report any co-changes for 12 out of 17 projects. We further discuss this lack of co-changes for MBA in Chapter 6.
In general, FO reported more co-changes than DTW did, except for the *RxJava* and *TestNG* projects. Aside from *Cassandra* and *PDFBox*, FO reported more than 1% of all pairs to be co-changing, whereas DTW only reported 6 projects above 1%.

5.1.1 Overlap

Only three projects of those analyzed had an overlap between algorithms of more than 5% of the union of the algorithms, which can be seen in Figure 5.1. it can be concluded that overlap is rare and the algorithms generally return significantly different co-changes.

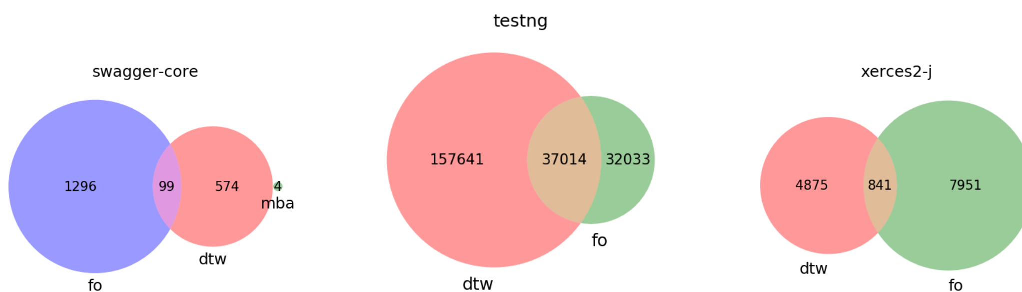

FIGURE 5.1: Overlap of output of the algorithm on three projects.

5.1.2 Co-changes over time

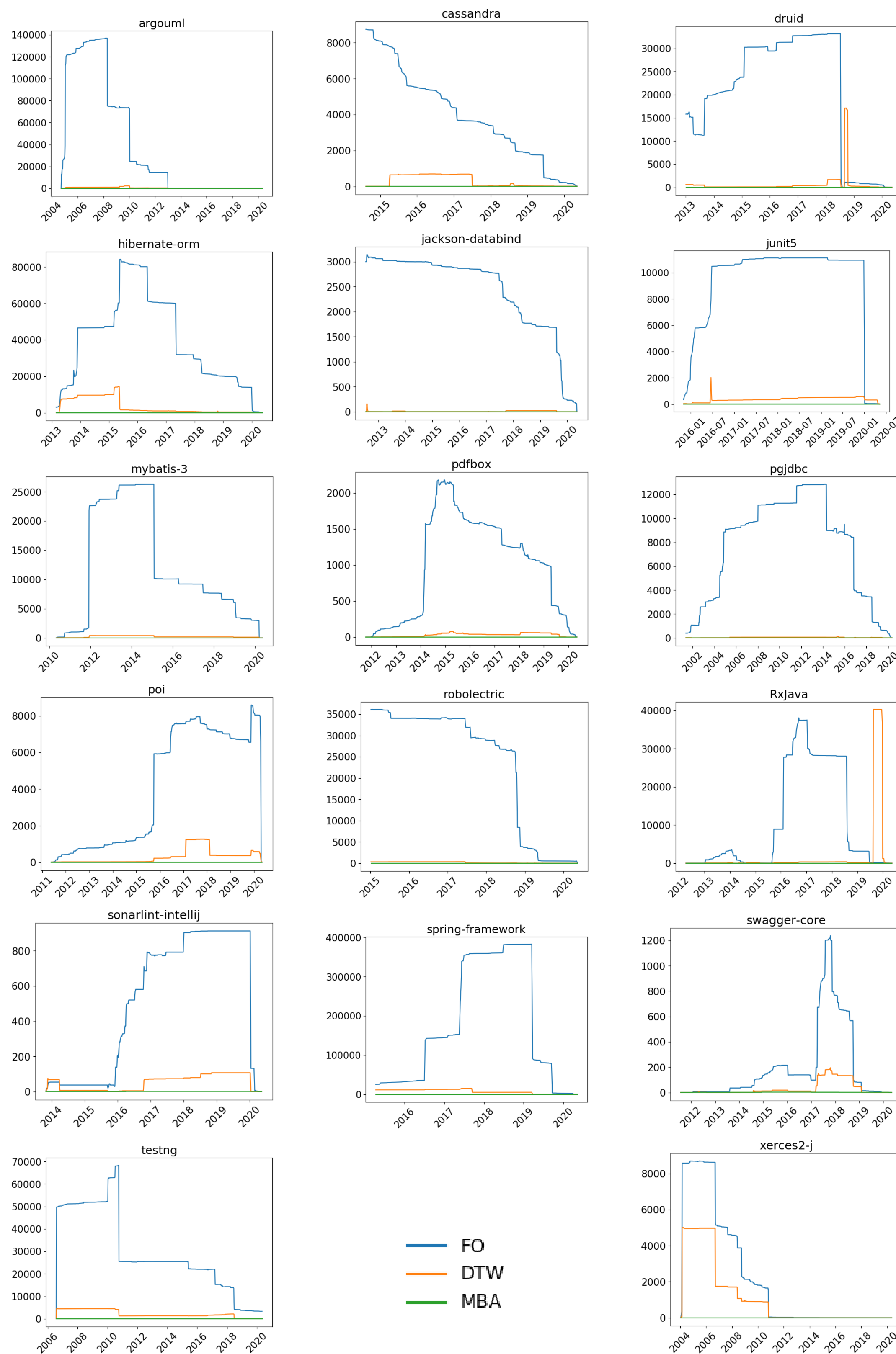

FIGURE 5.2: Co-changes over time.

Some interesting information can be gleaned from Figure 5.2. Visible in the graphs is that the development of *ArgoUML* and *Xerces2-j* was halted during the analysis period, due to which no co-changes were reported after the clear cut-off. While the algorithms have significantly different graphs, certain things are notable regardless of the algorithm. Some notably steep drops and climbs can be seen in the graphs when the number of co-changes greatly changes. Looking at the commits close to the day on which these notable changes occurred, it becomes noticeable that these changes are caused by considerably large commits, in which a significant number of files gets moved, added or deleted. Common occurrences are renames of folders or packages or restructurings of the repository.

The differences between the algorithms become notably more pronounced when looking at the co-changes over time. The lack of MBA results is clearly visible in the graphs, although some interesting patterns can be seen as well. In general, FO reports co-changes over a wider timespan, whereas with DTW, shorter 'spikes' of co-changes occur more often. However, in a number of graphs, it is clear that both algorithms report the same patterns and peaks in the number of co-changes, even if these patterns differ in magnitude.

5.2 RQ2 - Are artefacts affected by architectural smells co-changing?

In this section the results are given for RQ2. The percentage of relevant smelly pairs (i.e. falling in the right smell-scope) that is also co-changing is given per project and per algorithm. MBA is left out of the results because there were too few co-changes reported.

5.2.1 Results for Class AS

FO

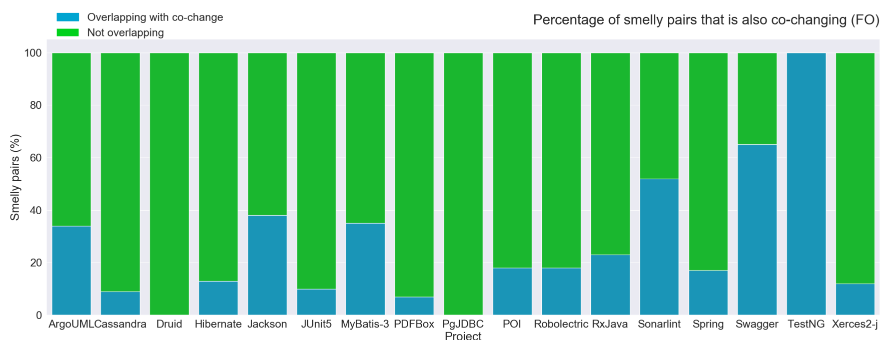

FIGURE 5.3: File pairs affected by class-level smells also reported as co-changing by FO.

Figure 5.3 shows the overlap for class-level smells using FO co-changes per project. On average, **26%** of relevant smelly pairs was also reported as co-changing per

project by FO. Figure 5.3 demonstrates that the actual percentage per project varied greatly. *TestNG*'s class-level smells are fully contained in its FO co-changes (100% overlap) but for *Druid* and *PgJDBC* no overlap was found. As it turns out, these are three projects with some of the lowest amounts of class-level smells; *TestNG* contains just one relevant smelly pair, and *Druid* and *PgJDBC* respectively contain 19 and 66. The average number of smelly pairs is 2490. To account for outliers with few relevant smelly pairs, all project datasets were combined into one and the overlap of the newly combined dataset was determined. **This yielded a *weighted average* of 15% when weighted by the number of relevant smells per project**.

DTW

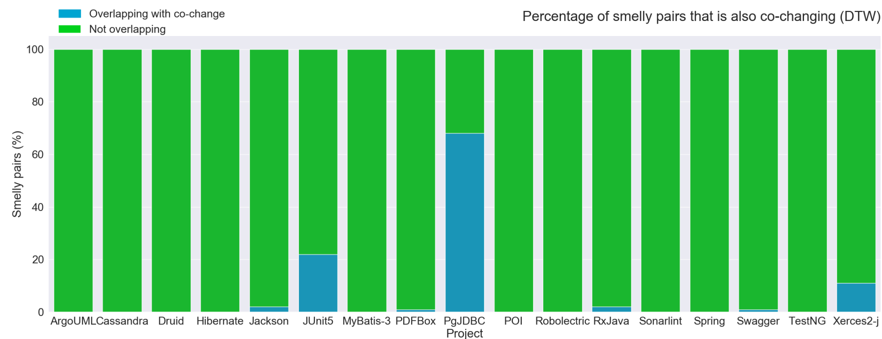

FIGURE 5.4: File pairs affected by class-level smells also reported as co-changing by DTW.

The results for DTW and class-level AS differed from the ones of FO. Figure 5.4 exhibits there was hardly any overlap in most projects, except for a few outliers. *PgJDBC* had an overlap of over 60%, although this still may have been caused by the small amount of smelly pairs (66). The same situation is true for *JUnit5* , containing only 49 relevant smelly pairs. *Xerces2-j*, on the other hand, contains 1910 smelly pairs and still reports a more significant overlap of 11%. In total this came down to an **unweighted average of 6%** and a **weighted average of 1%** of overlapping smells and DTW co-changes.

5.2.2 Results for Package AS

FO

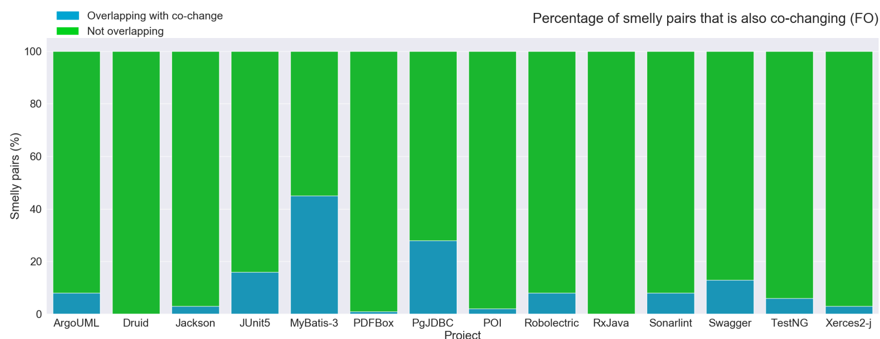

FIGURE 5.5: File pairs affected by package-level smells also reported as co-changing by FO.

The ratio of package-level smells matching with FO co-changes was significantly lower than for class-level smells, as can be seen in Figure 5.5. Most projects reported around 10% overlap, with a few exceptions. *JUnit-5*, *MyBatis-3* and *PgJDBC* again contained relatively low amounts of relevant smelly pairs: between 9,000 and 26,000, whereas the overall average was around 232,000. *ArgoUML* on the other hand contained more than 1,000,000 smelly pairs and still had an overlap of 8%. Overall, **an unweighted average of 10%** was found and **a weighted average of 5%**.

DTW

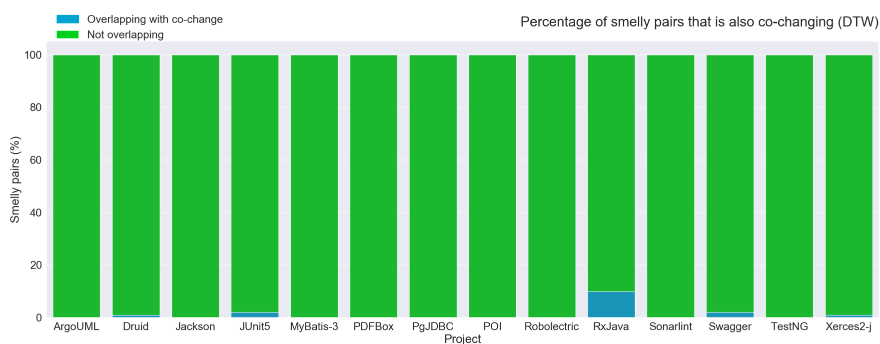

FIGURE 5.6: File pairs affected by package-level smells also reported as co-changing by DTW.

For DTW, the results again were lower than the respective ones produced by FO (Figure 5.6). For only one project the overlap reached the 10%: *RxJava*. This deserves mentioning as the project contains around 386,000 smelly pairs, which is high given the average of 232,000. Still, since this is the only project with such a percentage, **the unweighted and weighted average came out at 1%**.

5.2.3 Results for Class and Package AS

FO

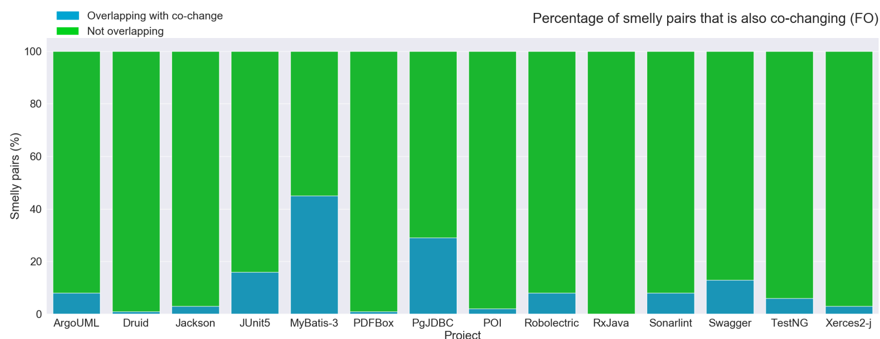

FIGURE 5.7: File pairs affected by class- or package-level smells also reported as co-changing by FO.

The results for the CP smell scope mostly follow those of the package-level scope for FO (Figure 5.7). However, since relatively more overlap was reported in the case of class-level smells, a slight increase in overlap can be noticed here compared to package-level only smells. **The unweighted average for this scope and FO was 10% and the weighted average was 6%**.

DTW

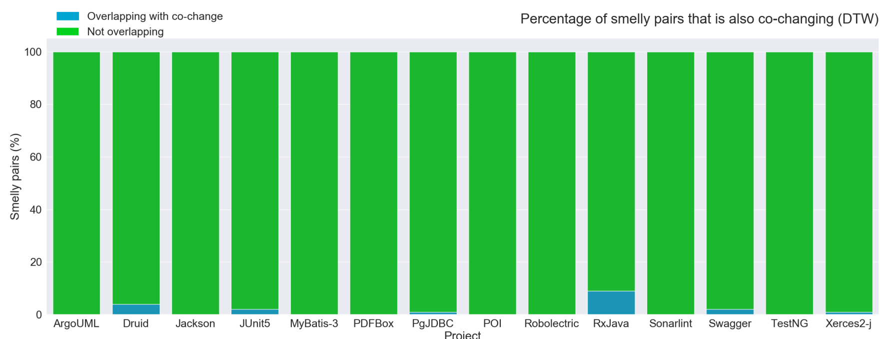

FIGURE 5.8: File pairs affected by class- or package-level smells also reported as co-changing by DTW.

Concluding the results of RQ2, Figure 5.8 paints a similar picture to Figure 5.6. Although DTW also reported slightly higher values for the class-level scope, the results for this scope still resemble the ones of the package-level: **an unweighted and weighted average of 1%**.

5.3 RQ3 - Are co-changes more often found in smelly artefacts?

The results of this RQ consist of the result of the χ^2-test and of the underlying required conditions to reject its null hypothesis. As mentioned before in Section 4.3, H_0^{RQ3} is rejected when the following conditions are met:

1. χ-value > 3.84
2. p-value < 0.05
3. ϕ-value > 0.1
4. o (odds ratio) > 1

The following tables capture these results, which are reported per algorithm and smell-scope. In the tables, a **bold** value indicates that the value passed the condition required to reject the null hypothesis. A dash (-) means that the required value could not be calculated. In the context of this RQ, this implies that one of the values of the contingency table of the χ^2 was 0; for example, all co-change pairs were also smelly, no co-changes were smelly, etc. Whenever a test could not be computed, no hypothesis is rejected nor accepted.

5.3.1 Result for Class AS

FO

Project	$H_0^{RQ3_FO_C}$	**χ-value**	**p-value**	**o**	**ϕ-value**
ArgoUML	Accepted	**9025.77**	**<0.01**	**9.99**	0.06
Cassandra	Accepted	**16342.81**	**<0.01**	**13.48**	0.08
Druid	-	-	-	-	-
Hibernate-ORM	Accepted	**2971.66**	**<0.01**	**6.88**	0.02
Jackson-databind	**Rejected**	**1282.91**	**<0.01**	**24.97**	**0.10**
JUnit5	Accepted	**4.46**	**0.03**	**3.06**	<0.01
MyBatis-3	Accepted	3.09	0.08	0.84	<0.01
PDFBox	Accepted	**2220.97**	**<0.01**	**10.90**	0.08
PgJDBC	-	-	-	-	-
POI	Accepted	**1812.87**	**<0.01**	**13.91**	0.05
Robolectric	Accepted	**169.62**	**<0.01**	**9.59**	<0.01
RxJava	Accepted	**2758.90**	**<0.01**	**5.93**	0.06
Sonarlint-IntelliJ	**Rejected**	**758.38**	**<0.01**	**30.10**	**0.16**
Spring-Framework	Accepted	**341.45**	**<0.01**	3.65	<0.01
Swagger-Core	**Rejected**	**2059.78**	**<0.01**	**71.98**	**0.20**
TestNG	-	-	-	-	-
Xerces2-j	Accepted	**417.93**	**<0.01**	**3.89**	0.04

TABLE 5.2: Results of testing H^{RQ3} with co-changes reported by FO and class-level AS. The threshold values for the four conditions can be found in Section 5.3.

Table 5.2 shows that the null hypothesis is rejected for **21%** of the projects. For the other projects, at least one of the conditions required to do so did not hold. This is mostly due to the required value of the ϕ-coefficient, which often is smaller than the required 0.1 and was the deciding factor for accepting the null hypothesis for 64% of the projects.

3 out of the 17 tested projects (18%) did not yield proper data for the test to be executed. 2 other projects (*MyBatis-3* and *JUnit5*) reported a minimal χ-value, one of which even lower than the critical value. For 4 out of 5, this can be explained by the lack of detected class-level AS: these projects reported between 1 and 100 file pairs affected by such smells. Most other projects reported between 1,000 and 20,000. However, this does not explain the results of *MyBatis-3*, which reported 442 smelly pairs, whereas *Sonarlint-IntelliJ* and *Swagger-Core* reported less and achieved χ- and ϕ-values orders of magnitudes higher. The reason for *MyBatis-3* is that it reported 26,581 co-changed file pairs, while *Sonarlint-IntelliJ* and *Swagger-Core* only reported 913 and 1,280. This explains why *MyBatis-3* scored low on the χ-test.

3 projects had a significant ϕ-value: *Sonarlint-IntelliJ*, *Swagger-Core* and *Jackson-databind*. These projects share certain properties within our dataset. Firstly, they contain a relatively small amount of files, causing them to be 3 out of the 5 smallest projects. Secondly, the amount of co-changes for them lies between 1000 and 3000 and the amount of smells between 100 and 250. Thirdly, the threshold applied when filtering FO co-changes is high for all three of them: respectively 12, 12 and 14. Although the threshold always results in roughly 5% of the most co-changing files of a project, a higher threshold might prevent more false positives.

So even though *there is a significant relation between co-changes and class-level AS in some projects*, in general **there appears to be no relation between class-level AS and FO co-changes**.

DTW

Project	$H_0^{RQ3_DTW_C}$	**χ-value**	**p-value**	***o***	**ϕ-value**
ArgoUML	Accepted	**25.52**	<**0.01**	0.07	<0.01
Cassandra	Accepted	**15.72**	<**0.01**	0.22	<0.01
Druid	-	-	-	-	-
Hibernate-ORM	Accepted	**88.85**	<**0.01**	0.01	<0.01
Jackson-databind	Accepted	**8.24**	<**0.01**	**4.78**	<0.01
JUnit5	Accepted	**274.76**	<**0.01**	**37.60**	0.03
MyBatis-3	-	-	-	-	-
PDFBox	Accepted	**171.30**	<**0.01**	**7.09**	0.02
PgJDBC	**Rejected**	**17240.34**	<**0.01**	**1522.98**	**0.36**
POI	Accepted	0.06	0.80	**1.02**	<0.01
Robolectric	-	-	-	-	-
RxJava	Accepted	**101.89**	<**0.01**	0.35	0.01
Sonarlint-IntelliJ	-	-	-	-	-
Spring-Framework	Accepted	**4.14**	**0.04**	0.35	<0.01
Swagger-Core	Accepted	0.06	0.81	**1.10**	<0.01
TestNG	-	-	-	-	-
Xerces2-j	Accepted	**628.35**	<**0.01**	**5.45**	0.05

TABLE 5.3: Results of testing H^{RQ3} with co-changes reported by DTW and class-level AS. The threshold values for the four conditions can be found in Section 5.3.

Table 5.2 shows that for **8%** of the projects the null hypothesis is rejected. For the other projects, at least one of the conditions required for this rejection did not hold. Interestingly, compared to the FO result, the average ϕ-value was lower yet only 7 out of 12 projects met the other three conditions compared to 12 out of 14 for FO as well. However, when the co-changes reported by FO did not match a single AS, *the null hypothesis for DTW was rejected for the project PgJDBC, with an average to large affect size*. Despite this, **we conclude class-level AS do not contain more DTW co-changes** as this conclusion is in line with 92% of the data.

MBA

As reported earlier, the MBA algorithm barely produced any co-changes for any of the analyzed projects. This means there is no overlap between AS and co-changes for this algorithm. We therefore had to accept $H_0^{RQ3_MBA_C}$ for all projects and concluded that **MBA co-changes do *not* occur more often in artefacts with class-level AS**.

5.3.2 Result for Package AS

FO

Project	$H_0^{RQ3_FO_P}$	**χ-value**	**p-value**	***o***	**ϕ-value**
ArgoUML	**Rejected**	**16505.71**	<**0.01**	**8.56**	**0.11**
Druid	Accepted	**515.35**	<**0.01**	**5.01**	0.03
Jackson-databind	-	-	-	-	-
JUnit5	Accepted	**1201.97**	<**0.01**	**2.78**	0.09
MyBatis-3	Accepted	**29.01**	<**0.01**	**1.10**	0.02
PDFBox	Accepted	**697.80**	<**0.01**	**56.18**	0.05
PgJDBC	**Rejected**	**1472.92**	<**0.01**	**2.37**	**0.16**
POI	Accepted	**2498.55**	<**0.01**	**11.20**	0.07
Robolectric	**Rejected**	**33703.90**	<**0.01**	**8.85**	**0.19**
RxJava	-	-	-	-	-
Sonarlint-IntelliJ	**Rejected**	**192.27**	<**0.01**	**3.75**	**0.12**
Swagger-Core	**Rejected**	**298.44**	<**0.01**	**3.02**	**0.14**
TestNG	**Rejected**	**18187.27**	<**0.01**	**3.76**	**0.11**
Xerces2-j	Accepted	**144.08**	<**0.01**	0.76	0.02

TABLE 5.4: Results of testing H^{RQ3} with co-changes reported by FO and package-level AS. The threshold values for the four conditions can be found in Section 5.3.

Table 5.4 shows that $H_0^{RQ3_FO_P}$ was rejected for **50%** of the projects. This is significantly higher than the percentage of rejections for class-level AS. For two projects, no smells that were not co-changing were present: *Jackson-databind* and *RxJava*. The deciding factor in the other projects was always the ϕ-value.

The common denominator between the projects with a significant ϕ appears to be that they have a high percentage of files that is affected by AS. For some projects this is true for over 75% of all files. In the same line, *RxJava* and *Jackson-databind* report so many AS that it would appear that all files are affected by these, making the χ^2-test impossible to calculate for these projects.

Based on these results, it is hard to draw a decisive conclusion as to answer RQ3. 6 out of 12 projects for which the analysis could be performed are shown to be related, some projects approach the values necessary for a relation (*JUnit5* and *POI*) while others are far from it (*MyBatis-3, Xerces2-j* and *Druid*). It can therefore be stated that **artefacts affected by package-level AS *might* contain more FO co-changes**.

DTW

Project	$H_0^{RQ3_DTW_P}$	**χ-value**	**p-value**	***o***	**ϕ-value**
ArgoUML	Accepted	**159.23**	<**0.01**	**5.11**	0.01
Druid	Accepted	**7885.66**	<**0.01**	0.16	**0.11**
Jackson-databind	-	-	-	-	-
JUnit5	Accepted	**389.84**	<**0.01**	**5.62**	0.05
MyBatis-3	Accepted	**22.52**	<**0.01**	**2.76**	0.02
PDFBox	Accepted	**358.28**	<**0.01**	0.15	0.04
PgJDBC	Accepted	**9.07**	<**0.01**	**1.99**	0.01
POI	Accepted	**156.94**	<**0.01**	**9.65**	0.02
Robolectric	Accepted	**19.42**	<**0.01**	0.63	<0.01
RxJava	Accepted	**136.20**	<**0.01**	0.64	0.02
Sonarlint-IntelliJ	Accepted	2.23	0.14	0.61	0.01
Swagger-Core	Accepted	2.32	0.13	**1.25**	0.01
TestNG	Accepted	**267.45**	<**0.01**	0.50	0.01
Xerces2-j	Accepted	**1157.93**	<**0.01**	0.32	0.07

TABLE 5.5: Results of testing H^{RQ3} with co-changes reported by DTW and package-level AS. The threshold values for the four conditions can be found in Section 5.3.

Although one relation was found on class-level AS, on package-level none were found for DTW. This can be explained by the smell and co-change ratio. As discussed for FO, some projects report an enormous amount of smelly artefacts (between 20,000 and 1,100,000). On the other hand, 10 out of 14 analyzed projects reported around 1000 co-changing pairs, which means that the overlap in terms of files was low. One clear exception to this rule is *Druid*, for which around 31,000 co-change pairs were reported, but for which the odds ratio was below 1. Generally speaking, we can conclude that **artefacts affected by package-level AS are *not* more likely to contain DTW co-changes**.

MBA

Because no overlap was found between co-changes and package-level AS in any of the 14 projects we accept $H_0^{RQ3_MBA_P}$ and conclude that **MBA co-changes are *not* more likely to be found in package-level smelly artefacts**.

5.3.3 Result for Class and Package AS

FO

Project	$H_0^{RQ3_FO_CP}$	**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-databind	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-IntelliJ	**Rejected**	**883.96**	<**0.01**	**6.11**	**0.17**
Swagger-Core	**Rejected**	**2944.03**	<**0.01**	**12.44**	**0.24**
TestNG	Accepted	**12252.85**	<**0.01**	**2.65**	0.08
Xerces2-j	Accepted	**8.40**	<**0.01**	0.94	<0.01

TABLE 5.6: Results of testing H^{RQ3} with co-changes reported by DTW and all AS. The threshold values for the four conditions can be found in Section 5.3.

When considering both types of smells, **50%** of the analyzed project passed all four conditions that needed to be met in order to reject $H_0^{RQ3_FO_CP}$. Furthermore, an extra 28% came close to the required ϕ-value and passed the remaining three conditions. Two projects stand out due to their relatively low ϕ-value: *Xerces2-j* and *Druid*, two projects of respectively average and large size. The results did not show a clear reason as to why the relation between AS and FO co-changes was so much weaker than for other projects. Since these two represented a mere 14% of the data, however, **it is likely there exists a relationship between AS and FO co-changes**.

DTW

Project	$H_0^{RQ3_DTW_CP}$	**χ-value**	**p-value**	***o***	**ϕ-value**
ArgoUML	Accepted	**5655.45**	<**0.01**	0.15	0.04
Druid	Accepted	**42.54**	<**0.01**	**1.42**	<0.01
Jackson-databind	Accepted	**620.30**	<**0.01**	0.05	0.07
JUnit5	Accepted	**86.55**	<**0.01**	**2.19**	0.02
MyBatis-3	Accepted	**28.94**	<**0.01**	**2.82**	0.02
PDFBox	Accepted	**106.66**	<**0.01**	0.34	0.02
PgJDBC	Accepted	**121.88**	<**0.01**	**4.13**	0.03
POI	Accepted	**340.64**	<**0.01**	0.42	0.02
Robolectric	Accepted	**41.38**	<**0.01**	0.55	<0.01
RxJava	**Rejected**	**26641.13**	<**0.01**	**5.76**	**0.17**
Sonarlint-IntelliJ	Accepted	<0.01	0.96	**1.02**	<0.01
Swagger-Core	Accepted	**6.83**	<**0.01**	**1.33**	0.01
TestNG	Accepted	**15129.81**	<**0.01**	0.04	0.09
Xerces2-j	Accepted	**830.31**	<**0.01**	0.39	0.06

TABLE 5.7: Results of testing H^{RQ3} with co-changes reported by DTW and all AS. The threshold values for the four conditions can be found in Section 5.3.

When relating DTW co-changes to AS of both levels, the results in Table 5.7 demonstrate that $H_0^{RQ3_DTW_CP}$ was rejected for only one project (**7%**). For most other projects, the null hypothesis was accepted with a low (< 0.05) ϕ-value and low odds-ratio (< 1). This implies that **there is no increased chance of co-changing in artefacts affected by AS, considering DTW co-changes**.

MBA

Although there was enough data to perform the χ^2-test for 2 projects (PgJDBC and Swagger-Core) both tests failed all four conditions and these results therefore were left out of this section. The other projects again did not contain any overlap between co-changes and AS, hence **we accept** $H_0^{RQ3_MBA_CP}$.

5.3.4 Summary

This section presents a summary of the results presented in the previous sections.

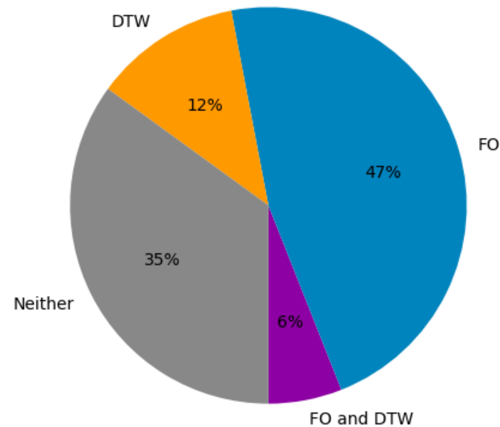

FIGURE 5.9: Percentages of projects for which at least one relation between smells and co-changes was found.

Figure 5.9 shows the amount of projects that are somehow related related to AS. **In 65% of the analyzed projects, certain co-changes are more often found in certain smelly artefacts**. However, the nature of this relationship varies per project. For most of these relationships, package-level smells are the major correlator. However, for certain smaller projects only class-level AS contain more co-changes. Most relations were found between FO co-changes and AS. DTW co-changes are only related to AS in 18% of all cases. No relation was found between MBA co-changes and AS.

5.4 RQ4 - Are smells introduced before or after files start co-changing?

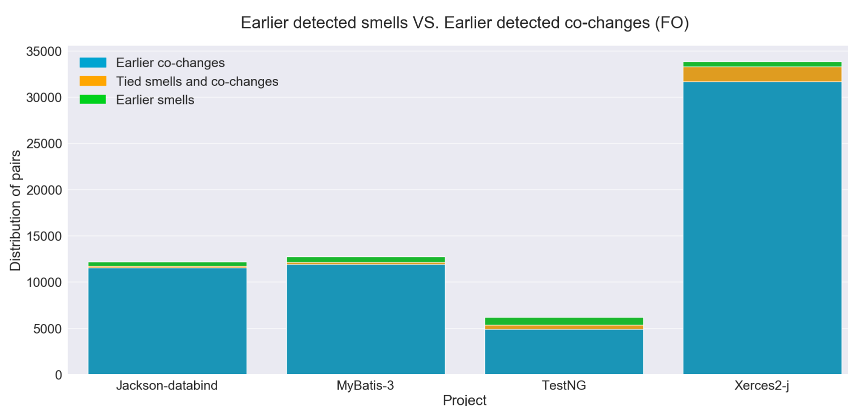

FIGURE 5.10: FO Results for projects with a large amount of overlapping file pairs.

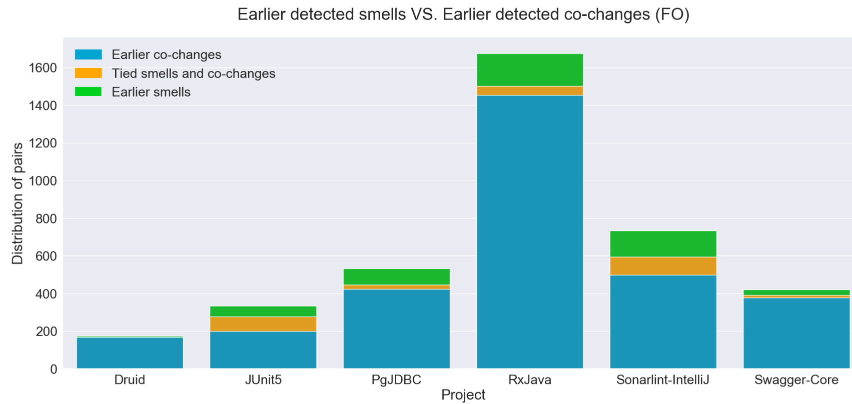

FIGURE 5.11: FO Results for projects with a smaller amount of overlapping file pairs.

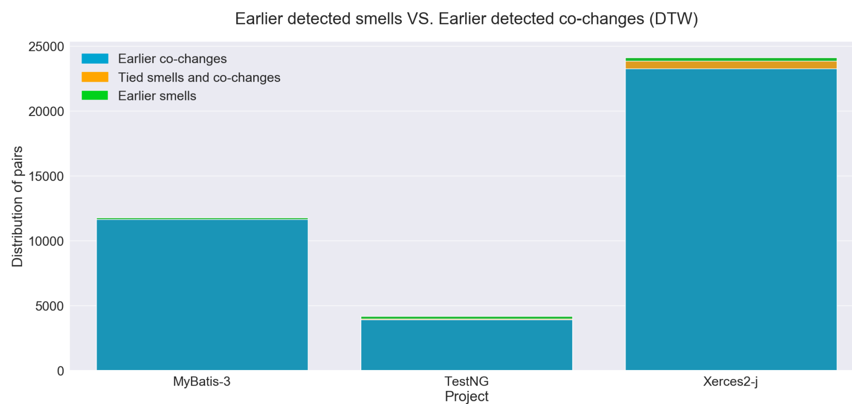

FIGURE 5.12: DTW Results for projects with a large amount of overlapping file pairs.

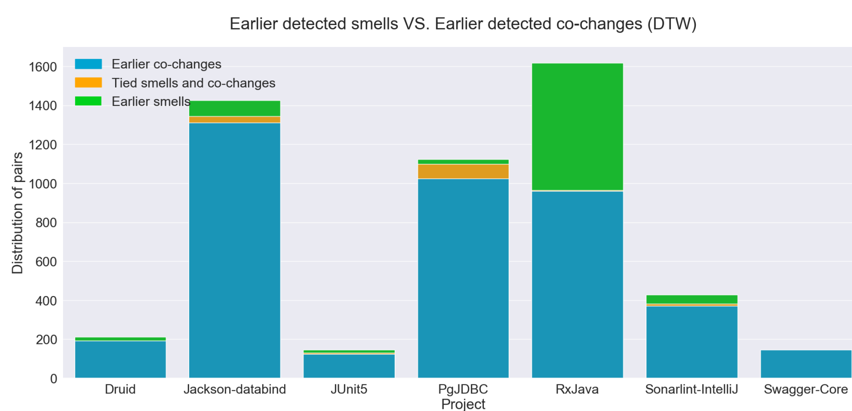

FIGURE 5.13: DTW Results for projects with a smaller amount of overlapping file pairs.

Figures 5.10, 5.11, 5.12, 5.13 show the results collected for RQ4. They demonstrate the amount of file pairs per project in case of which the smell was introduced before they started co-changing (earlier smell), for how many this was not the case (Earlier

co-change) and finally for how many they were introduced at the same time (Tied smells and co-changes).

The results for RQ4a and RQ4b are straightforward: for 100% of the tested projects and algorithms, $RQ4a_0$ was accepted, whereas $RQ4b_0$ **was rejected for 100% of the analyzed projects and algorithms**. All p-values were minimal ($<< 0.01$), regardless of which project and algorithm was considered. As shown by the figures, the difference between the two counts is significant: nearly all projects featured multitudes more earlier co-changes than smells. The amount of ties was always significantly lower than the amount the earlier co-changes, although it was higher than the amount of earlier smells for certain specific projects and algorithms (e.g. *PgJDBC* using DTW and *JUnit5* using FO). One result stands out: *RxJava* using DTW, which contained 654 earlier smells and 961 earlier co-changes. For the exact numbers per project and algorithm, please refer to Appendix A.

The results for MBA are not included in this chapter. The reason for this is, once again, the minuscule amount of reported co-changes. For four projects, results were gathered in the first place, but after examining the data more closely these were deemed meaningless.

Chapter 6

Discussion

This chapter discusses and interprets the results presented in Chapter 5 in more detail.

Comparing Co-change Detection Algorithms

The results obtained from RQ1 show a clear distinction between the co-change mining algorithms. Although related research has already applied the MBA and DTW algorithms for mining co-changes [47, 32, 9, 24], this study provides the first comparison of them with respect to mining co-changes. Furthermore, they were compared to a new algorithm, FO, which was specifically developed for this research.

Given that the co-changes in each of the three approaches barely overlapped (only incidentally above 5%) and that the reported co-change density also radically differed over time, it is implied that *the choice of co-change mining algorithm highly impacts which co-changes are reported*. The quality of the reported co-changes is hard to determine, so it is premature to express a value judgment regarding which algorithm would be preferred over the others.

Besides comparing output, the results of RQ1 gave insight in the configuration of the different algorithms. As repeatedly mentioned before, MBA offered no real results to speak of. This implies that the suggested configuration of Bavota et al. [8] (confidence $\geq$ 80%, support $\geq$ 2%) is not suitable for samples as large as the ones used in this case study. In general, the two thresholds given are too strict to make it possible to report co-changes. Which rule is the limiting factor depends on each project individually. Some projects contain many small commits, making the support the bottleneck (this is also suggested by Bavota et al.). For other projects, files are modified frequently but not relevantly. Frequently fixing typos or code style in singular files requires a lower confidence. *The takeaway here is that different projects require different configurations*.

The configuration of DTW also requires more attention if it is applied to co-change mining. This is implied by the results of RQ1, specifically Figure 5.2. This figure demonstrates large differences in certain projects' co-changes over time that were reported by DTW, which can likely be attributed to the configuration.

The threshold used by DTW is fixed for all projects, even though their histories take different shapes. Although DTW normalizes for the amount of steps (commits), it does not for the average distance (time) between commits. Concretely, this means that *different projects require different DTW thresholds*. Bouktif et al. [9] fine-tuned

the threshold for a specific project and our data suggests project-specific fine-tuning is indeed required. Moving on to how these different co-changes relate to AS, in the past other research has already related co-changes to different kinds of CPI, but not to AS. The data collected for this research has shown that a correlation between AS and co-changes likely exists, but that it is only small in effect.

Overlap Between Architectural Smells and Co-changes

This can be concluded from the results of RQ2 and RQ3. RQ2 served as an exploration for RQ3 in that it indicated the overlap between AS and co-changes of different algorithms; for FO, between 5 and 15% of smelly pairs was also found to be co-changing and for DTW around 1%. *This indicates that FO co-changes are more strongly related to AS, when compared to DTW or MBA*, the latter of which did not yield enough co-changes.

Besides reestablishing the clear difference in co-change algorithms, these results shed light on co-changing among smelly pairs: *smelly pairs are occasionally co-changing*. However, this does not imply any relationship between the two, since the percentage should be compared to the overlap of non-smelly pairs as explored in RQ3.

These results do require an important side note. The difference in overlap between the different smell-scopes shows that *the mapping between package-level smells and file pairs requires more fine-tuning*. For class-level smells, the overlap is three times higher than for package-level smells. This might be connected to the fact that all file pair combinations in a package are marked as smelly when, for example, a package-level's cyclic dependency only effects a few files in the relevant packages. When only the actually affected files in a package are considered smelly, this drastically decreases the amount of smelly pairs per project. Of course this does not guarantee more overlap, but it might result in more realistic data.

Earlier research established a relationship between some CPI and change-proneness [25, 26] and our research continues to explore this by looking for a correlation between AS and co-changes. The results of RQ3 show that *in a significant percentage (50%) of projects a correlation exists between FO co-changes and AS*. However, the effect size is rather small, but that may well be attributed to the large amount of package pairs, which heavily impacts the ϕ-value.

One implication of these results is that *AS only partially answer the question regarding the origin of co-changes*. This follows from the small effect size of the reported relationships between AS and co-changes and from the lack of relationships in some projects. In certain projects, AS might play a big role in co-changing, but in others it could be a different CPI: Knomh et al. have shown that artefacts containing code smells[25] and anti-patterns[26] are more likely to change, indicating these CPI could also be related to co-changing. Such other potential causes of co-changing might explain why no relationship with AS was found for certain projects in this research.

A second implication of these results is that *AS results in higher maintenance effort for a software project*. AS are considered an example of TD and as long as they

are not resolved, TD interest is being paid. This research shows that co-changes are one manifestation of this interest being paid, as smelly artefacts were found to co-change more often when compared to artefacts unaffected by AS. When files are co-changing, it implies that changing one file, often requires changing the other as well, which means more work has to be done. This demonstrates the potential increase in maintenance effort AS can bring.

Co-changes Before Architectural Smells

The goal of RQ4 was to look for a temporal connection between AS and co-changes, i.e. to see if one tends to precede the other. Such a connection was indeed reported by the data, but although the results appear straightforward, we refrain from drawing conclusions from them.

The major caveat here, which is why we do not draw conclusions, is found in determining the beginning of the time ranges of co-changes. While AS are clearly defined structures that can either be present or not, co-changes are not as well defined. Even if it is known when two files change together, it is still not known if this happens because of chance or because of some (hidden) internal dependency. *Without looking at the internal structure of two changes, we cannot know whether a co-change is merely a coincidence*. For all three algorithms used in this research, the first overlapping change of the two files was chosen as the start date of the co-change. This is the best the algorithms can do without looking at the internal change, but there is no guarantee this is correct. It could be that the two files were just added in the same commit. One coincidental overlapping change early in a project's history is enough for a pair to 'start co-changing' before it is affected by a smell, rendering the results of RQ4 potentially unreliable.

If the results are correct in the sense that co-changing indeed precedes AS, this would present some interesting implications. Since co-changing is typically a symptom of a CPI, it might be the case that a lower-level CPI precedes a higher-level AS, yet still generates the result of files co-changing, which would imply a shared cause. Another possibility is that co-changes themselves cause AS. If three artefacts always change together, it might feel intuitive and harmless to introduce a cyclic dependency or some other form of coupling among them, since a relationship already exists. Such implications, however, would require more research.

Although there is still much uncertainty regarding the relationship between AS and co-changes, the results of this research demonstrate that a link between them certainly exists and that smelliness in artefacts increases the chance of co-changing. This implies that there are indeed negative consequences to introducing TD (in the form of AS) with regards to maintainability. Moreover, the results show that co-changing can be an indication of future AS. Untangling co-changing files in time might therefore avoid the introduction of extra TD.

Chapter 7

Study Limitations

In this section, the limitations and threats to validity of the study are discussed as described by Runeson et al. [36]. These are split up in terms of *reliability*, *external validity* and *construct validity*. As we did not look at causal relationships, *internal validity* is not relevant to this study [36].

7.1 Construct Validity

Construct validity reflects to what extend the study measures what it is claiming 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. [36], 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.

However, we did identify certain threats to the construct validity. The first can be found in the detection of file changes. `CoCo` is capable of detecting file additions, moves and modifications. However, it is not capable of detecting when a file is deleted. This means that certain changes will not be detected by the application, which equally impacts the number of co-changes detected by all three algorithms. The impact of this threat is mitigated by its very nature. Deletes are rare compared to modifications, meaning that their impact on the co-change detection is relatively small. In addition, a deleted file can never change afterwards, reducing the chance that it will become a part of a co-changing pair.

Secondly, the choice of having ASTracker analyze each day instead of each commit separately could mean that information from certain other commits on the same day is lost. However, as ASTracker is only used to detect architectural smells, we consider the likelihood of this threat having impact on the results small. This is because architectural smells are unlikely to appear and disappear within the same day. Therefore, ASTracker will most likely correctly pick up all architectural smells, with only a small loss in accuracy as to when exactly the smell starts and stops appearing.

The third threat is found when co-changing files and files affected by an architectural smell are matched (i.e. finding OP). This match is performed based on the filename only, with no regards to package or directory. In order to prevent duplicates, only unique file pairs are kept. Therefore, if files with the same name exist within different

locations, information will be lost. However, this information is only lost if the equally named files co-change with the same files, as the duplicate pairs will then be filtered. If they co-changed with different files, no information would be lost. Duplicate filenames appear to be rare and in the projects selected in this study, they were only found in the *Sonarlint* project.

The next threat is related to the mapping of architectural smells to file pairs. For class-level smells, the overlap is three times higher than for package-level smells. This can be caused when all file pair combinations in a package are marked as smelly, even though, for example, a package-level cyclic dependency only affects a few files in the relevant packages. The current approach is not indisputably wrong, as some package-level smells do affect all files in the package, but further research into the correct way of converting package-level smells to file pairs is needed.

In the current study design, the co-change threshold (see Subsection 3.4.1) is set to the 95th percentile. Using a consistent percentage means that we ignore differences between projects, as a different percentile could be needed for different projects. The current static percentile is not a great threat in and by itself, but it could cost us some nuances in the results that could be of interest. However, the fact that the percentile is calculated per project instead of once for all projects mitigates some problems.

The start and end dates of a co-change also pose a possible threat. 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, which can nevertheless be partially mitigated if the threshold percentile filters out file pairs that do not change often enough.

The final threat to construct validity is found in the implementation of the FO algorithm. The comparison and the results depend on the algorithm being correctly implemented. In order to guarantee that our implementation is close to optimal, we have developed system and unit tests to verify the results produced by `CoCo`[1]. Manual verification of detected outliers was also performed to verify correctness.
However, even that comes with its own footnotes. As mentioned in Chapter 6, we have no way of verifying the quality of the reported co-changes, as co-changes lack a clear definition. This makes it challenging to express a value judgment with regard to the output of the analyzed algorithms, without further in-depth research.

7.2 External Validity

External validity is a reflection of how well the results of this study can be extended to other projects, given a similar context.

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 generalized to other open source projects, and not necessarily to other kinds of projects. In addition, 5 out of 17 projects are owned by the Apache Foundation, which impacts the generalization of results to other authors. We have, however, made sure to mitigate

[1] `https://github.com/RonaldKruizinga/CoChangeDetectTest`

this by choosing projects from various domains, from which we identified 5 specific domains, each with a similar number of projects.
The second threat is related to the size of projects. Care was taken to choose projects with various repository sizes and commit histories of varying lengths. However, the larger projects with longer histories, such as *Cassandra*, *Hibernate* and *Spring* proved too large to analyze with the current application and hardware, due to which only RQ1 and parts of RQ2 and RQ3 could be answered for these projects. Therefore, the results of the other research questions cannot be generalized to these large projects with long histories.
The third threat is regarding the specific architectural smells that were chosen to analyze. It is incredibly difficult, if not impossible, to generalize the results unto other architectural smells as the results greatly depend on the type of smell and its detection strategy.

7.3 Reliability

Reliability concerns 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. Hyperparameters for all projects analyzed are also available in Appendix B. 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 discussed by the thesis authors and discussed on a weekly basis with the thesis supervisor in order to ensure reliability.

In addition, similar data collection and analysis techniques have been used in previous studies on architectural smells [38] and co-change detection [8, 9], assuring that such an approach to the analysis of these artefacts is possible.

Chapter 8

Conclusion

This research has intensively investigated co-changes and their relation to architectural smells (AS). A case study was set up analyzing 17 open source projects and an accumulated 20,000 changesets (commits), capturing decades of software change history. From this dataset, file pairs were mined that changed synchronously: co-changes. Three different algorithms were used for this: two state-of-the-art approaches (Dynamic Time Warp and Market Basket Analysis) and a new algorithm that intends to combine the best of the two worlds: Fuzzy Overlap (FO).

This new FO algorithm was implemented in `CoCo`, a publicly available application that reports changes and co-changes in Git projects. This data was combined with AS information generated by two other tools: Arcan and ASTracker.

This dataset was then explored and statistically analyzed from several perspectives. The results of this elaborate analysis yielded the first comparison of co-change mining approaches (1), showing that the algorithms report highly diverse co-changes and that their configuration is paramount to getting the right data.

The results have also shown that co-changes are found more often in certain smelly artefacts (2), indicating that AS increases maintenance effort in certain projects. The temporal relationship between AS and co-changes was also investigated (3) but, although the findings appeared unanimous, no conclusions could be drawn from this due to the still complex nature of co-changes.

The complexity of co-changes and their underlying causes introduced challenges during the study and certain remarks are still required to be made regarding the methods and results. Package-level AS has turned out to be challenging to properly relate to co-changes and the influence of the chosen algorithm configurations remains questionable.

After all, however, this research has still given a broad overview of co-changes and how to mine them and has found that, in extension to lower-level smells, AS are in fact related to co-changes.

8.1 Future Work

For future research, several improvements to this study could be made. The threats to validity discussed in Chapter 7 should be tackled by, for example, further research into the proper co-change threshold and the best way to map package-level AS to file pairs.

To improve on the FO algorithm, the relationship between several commit and project attributes should be investigated. In particular, co-change detection could take into account the author of a commit, since commits from different authors could be less likely to be part of a co-change. The size of a file could also be taken into account when looking at co-changes, since a file with more lines of code could be more likely to change in general, impacting how often it gets reported in a co-changing pair.
A possible relationship with the domain of a project would also be interesting to look into. Different domains could have a different change patterns and thus require different hyperparameters.
A final thing to look into for the purpose of improving FO is the existence of exceedingly large commits. Most projects have a commit in which most, if not all, files are moved to a different directory. This happens, for example, when a major release is completed or a major refactoring is done. This causes all files to change together with every other file, which has a large impact on the start and end dates of a co-change. It would be interesting to investigate whether these should be excluded from the file history, similar to merge commits.

Further research into proper thresholds for the MBA and DTW algorithms could also prove useful. Choosing the right thresholds for those algorithms could be done dependent on the project to be analyzed.

More work could also be done regarding the architectural smells analyzed. Only 3 types are currently reported by ASTracker, yet many more exist. Analyzing more smells could offer more insight in the relationship between AS and co-changes.
Similarly, more projects should be analyzed in order to increase the sample size. It has become clear that there exist large differences between certain projects, due to which some interesting insights could surface when more projects are analyzed. In order to properly support this hypothesis, the analysis should be improved as to no longer be limited due to memory constraints.

Finally, RQ4 implies that CC and AS could share a common cause, since for all projects it held that CC precede AS. This could prove an interesting topic for further research into the origins of both CC and AS.

Appendix A

RQ4 Results

This appendix contains the tabulated data collected for Section 5.4.

A.1 RQ4a Results

A.1.1 FO

Project	H_0	Smell < CC	Smell ≥ CC	p-value
Druid	Accepted	8	166	<<0.01
Jackson-databind	Accepted	450	11718	<<0.01
JUnit5	Accepted	58	276	<<0.01
MyBatis-3	Accepted	600	12158	<<0.01
PgJDBC	Accepted	87	446	<<0.01
RxJava	Accepted	173	1502	<<0.01
Sonarlint-IntelliJ	Accepted	139	595	<<0.01
Swagger-Core	Accepted	29	391	<<0.01
TestNG	Accepted	850	5360	<<0.01
Xerces2-j	Accepted	565	33285	<<0.01

A.1.2 DTW

Project	H_0	DTW_earlier_smell_pairs	DTW_cc_or_tied	DTW_p_smell_first
Druid	Accepted	18	193	<<0.01
Jackson-databind	Accepted	82	1344	<<0.01
JUnit5	Accepted	15	131	<<0.01
MyBatis-3	Accepted	148	11637	<<0.01
PgJDBC	Accepted	25	1099	<<0.01
RxJava	Accepted	654	964	<<0.01
Sonarlint-IntelliJ	Accepted	47	381	<<0.01
Swagger-Core	Accepted	0	146	<<0.01
TestNG	Accepted	198	3983	<<0.01
Xerces2-j	Accepted	264	23851	<<0.01

A.1.3 MBA

Project	H_0	MBA_earlier_smell_pairs	MBA_cc_or_tied	MBA_p_smell_first
Druid	-	-	-	-
Jackson-databind	-	-	-	-
JUnit5	Accepted	0	7	0.016
MyBatis-3	-	-	-	-
PgJDBC	Accepted	0	165	<<0.01
RxJava	-	-	-	-
Sonarlint-IntelliJ	Accepted	3	249	<<0.01
Swagger-Core	Accepted	0	125	<<0.01
TestNG	-	-	-	-
Xerces2-j	-	-	-	-

A.2 RQ4b Results

A.2.1 FO

Project	H_0	FO_earlier_ccs_pairs	FO_smell_or_tied	FO_p_cc_first
Druid	**Rejected**	166	8	<<0.01
Jackson-databind	**Rejected**	11533	635	<<0.01
JUnit5	**Rejected**	198	136	<<0.01
MyBatis-3	**Rejected**	11922	836	<<0.01
PgJDBC	**Rejected**	422	111	<<0.01
RxJava	**Rejected**	1453	222	<<0.01
Sonarlint-IntelliJ	**Rejected**	499	235	<<0.01
Swagger-Core	**Rejected**	378	42	<<0.01
TestNG	**Rejected**	4882	1328	<<0.01
Xerces2-j	**Rejected**	31681	2169	<<0.01

A.2.2 DTW

Project	H_0	DTW_earlier_ccs_pairs	DTW_smell_or_tied	DTW_p_cc_first
Druid	**Rejected**	193	18	<<0.01
Jackson-databind	**Rejected**	1310	116	<<0.01
JUnit5	**Rejected**	123	23	<<0.01
MyBatis-3	**Rejected**	11633	152	<<0.01
PgJDBC	**Rejected**	1024	100	<<0.01
RxJava	**Rejected**	961	657	<<0.01
Sonarlint-IntelliJ	**Rejected**	371	57	<<0.01
Swagger-Core	**Rejected**	146	0	<<0.01
TestNG	**Rejected**	3911	270	<<0.01
Xerces2-j	**Rejected**	23238	877	<<0.01

A.2.3 MBA

Project	H_0	MBA_earlier_ccs_pairs	MBA_smell_or_tied	MBA_p_cc_first
Druid	-	-	-	-
Jackson-databind	-	-	-	-
JUnit5	**Rejected**	7	0	0.016
MyBatis-3	-	-	-	-
PgJDBC	**Rejected**	165	0	
RxJava	-	-	-	-
Sonarlint-IntelliJ	**Rejected**	244	8	<<0.01
Swagger-Core	**Rejected**	125	0	<<0.01
TestNG	-	-	-	-
Xerces2-j	-	-	-	- <<0.01

Appendix B

Hyperparameter Analysis Results

Project	Startdate[1]	Co-change Threshold	Total number of commits to analyze	Commits Analyzed[2]	Time between commits in hours	Commits between commits
ArgoUML	17/09/2004	6	12000	1661	5	6
Cassandra	14/08/2014	15	11000	2089	2	9
Druid	02/01/2013	4	10000	1591	6	6
Hibernate	27/02/2013	3[3]	6000	2142	8	5
Jackson	18/07/2012	14	6000	1173	10	5
JUnit5	31/10/2015	16	6039	884	3	6
MyBatis-3	10/05/2010	5	3339	992	22	3
PDFBox	13/10/2011	6	8000	1652	11	4
PgJDBC	16/05/2001	9	2538	1048	65	3
POI	20/05/2011	4	6000	1322	14	4
Robolectric	06/01/2015	14	5796	979	6	5
RxJava	10/04/2012	7	5734	1231	7	5
Sonarlint	30/10/2013	12	1200	290	20	4
Spring	12/04/2015	5	10743	1250	4	7
Swagger	28/07/2011	12	4000	884	12	4
TestNG	30/07/2006	4	5000	1001	19	4
Xerces2-j	22/01/2004	4	2000	860	15	4

TABLE B.1: Result of the hyperparameter analysis for all projects.

[1] All projects have an enddate of 14/05/2020.

[2] Note that this is always less than the total number of commits, since ASTracker only analyzes one commit per day.

[3] For this project the 96th percentile was taken, as the 95th percentile was the same as the 85th, which would cover too much of the data.

Bibliography

[1] N. Ajienka, A. Capiluppi, and S. Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 432–432, May 2018.

[2] F. Arcelli Fontana, P. Avgeriou, I. Pigazzini, and R. Roveda. A study on architectural smells prediction. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 333–337, 2019.

[3] F. Arcelli Fontana and S. Maggioni. Metrics and antipatterns for software quality evaluation. In *2011 IEEE 34th Software Engineering Workshop*, pages 48–56, 2011.

[4] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285, 2017.

[5] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni. Automatic detection of instability architectural smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 433–437, 2016.

[6] U. Azadi, F. Arcelli Fontana, and D. Taibi. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97, 2019.

[7] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk... In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, volume 11, 1997.

[8] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701, 2013.

[9] A. Bouktif, Y. Gueheneuc, and G. Antoniol. Extracting change-patterns from CVS repositories. In *2006 13th Working Conference on Reverse Engineering*, pages 221–230, 10 2006.

[10] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., USA, 1st edition, 1998.

[11] T. Z. C. Bird, T. Menzies. *The Art and Science of Analyzing Software Data*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[12] W. Cunningham. The WyCash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, page 29–30, New York, NY, USA, 1992. Association for Computing Machinery.

[13] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the size, cost, and types of technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 49–53, 2012.

[14] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[15] M. Fowler. Refactoring: Improving the design of existing code. In D. Wells and L. Williams, editors, *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, pages 256–256, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[16] M. Fowler. Codesmell. https://martinfowler.com/bliki/CodeSmell.html, 2006. Last accessed on 2020-07-03.

[17] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23, Sep. 2003.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[19] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.

[20] A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, 2009.

[21] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, 2015.

[22] IEEE. Standard for a software quality metrics methodology. *IEEE Std 1061-1992*, pages 1–96, 1993.

[23] ISO, IEC, and IEEE. International standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, 2010.

[24] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol. An exploratory study of macro co-changes. In *2011 18th Working Conference on Reverse Engineering*, pages 325–334, 2011.

[25] F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84, Oct 2009.

[26] F. Khomh, M. Di Penta, Y. Gueheneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, Jun 2012.

[27] C. Larman. *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2004.

[28] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. Relating architectural decay and sustainability of software systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 178–181, 2016.

[29] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.

[30] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60, 2015.

[31] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[32] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 103–112, 2013.

[33] M. Mondal, C. K. Roy, and K. A. Schneider. A fine-grained analysis on the evolutionary coupling of cloned code. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 51–60, 2014.

[34] M. Nayrolles, N. Moha, and P. Valtchev. Improving SOA antipatterns detection in service based systems by mining execution traces. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 321–330, 2013.

[35] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In *2008 15th Working Conference on Reverse Engineering*, pages 42–46, Oct 2008.

[36] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition, 2012.

[37] D. Sas, P. Avgeriou, and F. Arcelli Fontana. Investigating instability architectural smells evolution: An exploratory case study. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 557–567, 2019.

[38] D. Sas, P. Avgeriou, I. Pigazzini, and F. Arcelli Fontana. (under review) on the relation between architectural smells and source code changes. In *TechDebt*

2020: Proceedings of the Third International Conference on Technical Debt, 2020.

[39] D. J. Sheskin. *Handbook of PARAMETRIC and NONPARAMETRIC STATISTICAL PROCEDURES*. CHAPMAN & HALL/CRC, 2000.

[40] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. 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*, pages 300–310, 09 2011.

[41] I. Sommerville. *Software Engineering*. Pearson, 10th edition, 2016.

[42] H. van Vliet. *Software engineering: principles and practice*, volume 13. John Wiley & Sons, 2008.

[43] B. F. Webster. *Pitfalls of object-oriented development*. M & T Books, 1995.

[44] P. Weißgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 9–9, 2007.

[45] Z. Zhang, P. Tang, L. Huo, and Z. Zhou. MODIS NDVI time series clustering under dynamic time warping. *International Journal of Wavelets, Multiresolution and Information Processing*, 12(05), 2014.

[46] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR*, volume 4, pages 2–6, Los Alamitos CA, 2004. IEEE Press.

[47] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 563–572, USA, 2004. IEEE Computer Society.