SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of

Informatics Systems and Communication

PhD program Computer Science Cycle XXXIV

Evaluating and Detecting Architecture Erosion

Surname Pigazzini Name Ilaria

Registration number 780684

Tutor: Prof. Alberto Ottavio Leporati

Supervisor: Prof. Francesca Arcelli Fontana

Coordinatore / Coordinator: Prof. Leonardo Mariani

ACADEMIC YEAR 2020/2021

ABSTRACT

A software architecture is eroded (or degraded) if it shows a progressive loss of structural integrity due to design principle violations which leads to the deviation of the implemented architecture from the intended architecture [223]. Eroded systems suffer from Architectural Technical Debt (ATD), the additional effort required by developers to manage the shortcomings caused by the erosion. A symptom of the accumulation of ATD is the presence of Architectural Smells (AS), design decisions that impact negatively on the internal system quality. Systems affected by AS suffer from higher maintenance costs and are harder to evolve. This thesis investigates six different types of AS violating different design principles in Open-Source and industrial monolithic Java projects. We identify AS with our tool, Arcan, and introduce its new extension for the representation of software concerns. We then discuss AS from the point of view of practitioners, trying to summarise how AS are perceived and validating Arcan results. We also report the results of our empirical studies concerning AS and ATD evolution and correlation. Finally, we present our first results concerning the migration and maintenance of microservices architectures, with a focus on the detection of microservices smells.

Here we may reign secure, and in my choice
To reign is worth ambition though in Hell:
Better to reign in Hell, than serve in Heav'n.

— John Milton, *Paradise Lost*

And malt does more than Milton can
To justify God's ways to man.

— A.E. Housman, *Terence, This is Stupid Stuff*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

We live in a world heavily relying on software, where developing *good* software is of fundamental importance. However, what do we mean with "a good software"? In software engineering we agree with measuring the goodness of software by evaluating different software quality attributes [108], such as reliability, security and maintainability. In particular in this thesis we take into consideration the quality of *software architectures* and what happens when such quality is compromised. Software systems have large and complex architectures that are the result of ongoing design processes involving the decisions of several developers and architects. Starting from an intended architecture built upon planned design choices and following specific design principles, the evolution of the system may lead to the deviation from the original architecture, and the system may experience *architecture erosion*. A software architecture is eroded (or degraded) if it shows a progressive loss of structural integrity due to design principle violations which leads to the deviation of the implemented architecture from the intended architecture [223]. Erosion is a natural condition for an evolving architecture. Even without considering systems developed with agile practices [194], in general the software life-cycle includes progressive adaptations according to the possible changes in requirements, bug-fixing, changes in the execution environment and implementation upgrades, such as the substitution of a library with another. All these activities implicate the evolution of the architecture and the consequent deviance from the original architecture, often resulting in architectural violations [100]. However, if such violations are taken into account and fixed, and the software architecture evolves along with the system, i.e., the architecture is restructured according to the new requirements, erosion could be a temporary condition. On the contrary, if the architecture progressively erodes without containment actions (such as refactoring activities [240]), then erosion becomes a persistent problem impacting the system quality, in particular its maintainability, performance and capability of evolve [100].

In other words, this means that systems affected by architecture erosion can suffer from software performance decrease, for instance the systems could require more computational resources. Moreover, the system could be hard to maintain, meaning that a small bug-fixing that should require few minutes work, ends up in the refactoring of large portions of code [177]. Last but not the least, designing and implementing new functionalities could become a troublesome activity, because eroded systems are usually structurally entangled, hard to

comprehend and hard to integrate with new components [102]. That is why it is important to identify architecture erosion, so that developers can take action and remove it.

However, in order to manage architecture erosion, we need a mean to identify and quantify it. We mentioned that the cause behind the architecture erosion is the introduction of architecture violations. The amount of architecture violations in a system is called *Architectural Technical Debt* (ATD) [153], a sub-type of the wider concept named *Technical Debt* (TD). TD is a metaphor introduced in 1992 by Ward Cunningham [62]. TD is the consequence of bad design or implementation decisions which seems to provide benefit in the short term, but impact negatively on the future of the software. In this thesis we do not consider implementation decisions, but focus mainly on identifying and quantifying architectural debt, i.e., the design decisions related to architectural layers, subsystems, interfaces, technologies and frameworks, among the others [246].

It is not trivial to identify ATD in a system, because the causes of the debt cross many aspects related to the software life-cycle: debt can be introduced by mistake by developers, could be caused by the time-to-market pressure affecting developers, could be due to the lack of team expertise or the lack of architecture documentation [246]. That is why a part of the research focus on the study of ATD *symptoms*, i.e., hints about the presence of a source of debt: identifying a symptom is the first step for diagnosing the debt. The thesis focuses on this aspect: the study of ATD symptoms, their detection and evaluation.

A specific category of symptoms largely investigated in this thesis is the one of *Architectural Smells* (AS). An architectural smell is a commonly used architectural decision that negatively impacts the system internal quality [91], i.e., they are the result of the above mentioned architectural violations[105].

AS come in many types, depending on which design principle they violate. In this thesis, we focus our studies on six types of AS, divided into the three related to architecture dependency issues, namely Cyclic Dependency, Unstable Dependency and Hub-Like Dependency; the one affecting architecture modularity, named God Component; the two breaking the separation of (software) concerns, namely Scattered Functionality and Feature Concentration.

The first part of the thesis focus on Arcan: an automatic tool for architectural smells detection. We exploit Arcan to identify the six AS and collect data useful for statistical analysis on software evolution and quality. In the following chapters, we first describe Arcan components and detection strategies. A particular attention is given to the introduction to our approach for the modeling of architectural concerns and for the identification of the two smells violating the separation of concerns principle, Scattered Functionality and Feature Concentration. Following, we present our studies investigating the

impact of AS on software quality. To reach our goal, we probed the perception of practitioners, conducting a set of case studies in industrial context [79][217] and asking them in what ways AS impacted software quality according to their experience. At the same time, we seized the opportunity to validate the results of our tool, in terms of precision.

We then describe our empirical studies investigating the correlation and evolution of AS in Open-Source projects, starting from the analysis of the relationships between AS and Design Patterns (DP) [88]. Given that DP adoption is widely recommended, since they are verified and distilled design solutions, we could expect that AS and DP represent different concepts in terms of software quality and as such are mutually exclusive. However, we found out that these two concepts can be related in some cases [199], e.g., the occurrence of architectural smells can interfere with the presence of design patterns. Moreover, we report our study about the evolution and correlation among different types of AS: in this case we found that two smells named Cyclic Dependency and Unstable Dependency are often collocated, suggesting a possible common cause behind their introduction.

As already outlined, AS are a symptom of ATD, thus the identification of smells is crucial to identify this source of debt. Concerning this aspect, we describe our studies aimed to evaluate the ATD amount, measured in terms of AS, of Java projects belonging to different application domains. One way to quantify the ATD of a given project is to evaluate an *ATD index*, i.e., a numerical value that varies as long as ATD increases or decreases. We exploit our existing ATD index based on AS detection, named Architectural Debt Index (ADI), to compute the amount of ATD in our studies. In particular, we investigate the relationship of ADI evolution in Open-Source projects with the content of the commit messages written by their developers [80][196]. For instance we take in consideration messages indicating a bug-fixing, a code improvement or a refactoring, to understand whether such activities influence the ATD trend during the evolution of a project. Moreover, we report a study we participated in concerning the impact of opportunistic reuse practices on TD and ATD [50]. We also participated in a large study about technical debt tools [26], however we do not describe it in this thesis.

The instances of AS are not equally critical for the system. Indeed, one of the most challenging theme emerging from AS studies is AS prioritization, i.e., to order the smell instances depending on their criticality. Concerning this direction, we investigated the relationship among AS criticality and the PageRank of the system under analysis. PageRank is a measure that estimates whether an architectural smell is located in an important part of the project, where the importance is evaluated according to how many parts of a project depend on the one involved in the architectural smell. We report both the

preliminary case study we conducted about such topic [82] and the subsequent empirical study [200].

All the studies mentioned until now regarded smells affecting *monolithical* architectures, i.e., self-contained systems made of strongly coupled components. However, we also studied the role of architectural smells on the migration from monolithic architecture to *microservices* architecture, an architectural style where, differently from the monolithical one, the components (services) are loosely coupled. In particular, we introduced an approach to identify candidate microservices in monolithic Java projects by exploiting AS detection and we validated such approach in two case studies in industrial settings [198] [79].

Finally, we developed an extension of the Arcan tool and a new tool, named AROMA (Automatic Recovery of Microservices Architecture), for the detection of *microservices smells*. This kind of smells are the counterpart of AS in microservices architecture. In this case, we did not have the opportunity to conduct industrial case studies, however we report the first results of the analysis ran on Open-Source projects.

The main subject of this thesis, ATD and AS, is large and includes many different facets. Indeed, as just outlined, we addressed with our research many topics that concern AS occurrence, perception and evolution. For the sake of simplicity, for each study referring to an investigated problem (represented by each chapter), we indicate its research questions, and discuss the overall findings at the end of each chapter, with the aim to link the results one with another. Moreover, in Chapter 8 we discuss all the topics introduced in the thesis. In the chapter, we eventually answer a set of *recapitulatory* questions with the aim to summarise and give shape to the findings of the thesis.

In brief, the main contributions of this thesis are:

- Definition and implementation of two new detection algorithms for the identification of architectural smells violating the separation of concerns design principle - Chapter 2;
- Validation and analysis of the perception of architectural smells in industrial context - Chapter 3;
- Empirical studies of architectural smells correlation and evolution - Chapter 4;
- Architectural debt evaluation of Open-Source projects - Chapter 5;
- Study of the role of AS during the migration to microservices and definition and implementation of new algorithms for microservices smells detection - Chapter 6.

We aim with this thesis to concretely contribute to the research in the field of AS detection and ATD management. The development of dedicated algorithms and tools for AS detection can be of valid

support for developers facing the negative consequences of AS everyday. On the other hand, our studies about the correlation, evolution and impact of AS are useful for both practitioners and researchers, to acquire empirical knowledge about the nature of AS.

1.1 PUBLICATIONS

The lists of published and submitted papers are reported in the following. The contribution of Pigazzini is indicated at the end of the reference. Papers covered in this thesis are marked with [*Discussed*] and the name of the corresponding section.

1.1.1 *Published papers*

Darius Sas, Paris Avgeriou, **Ilaria Pigazzini**, Francesca Arcelli Fontana, *"On the relation between architectural smells and source code changes"* in Journal of Software: Evolution and Process (JSEP), 2021 (in press). Contribution: performed part of the analysis.

Darius Sas, **Ilaria Pigazzini**, Paris Avgeriou and Francesca Arcelli Fontana, *"The perception of Architectural Smells in industrial practice"* in IEEE Software, August 2021 (Early Access). Contribution: collected the data, performed the analysis and wrote part of the paper. [*Discussed*, Section 3.3]

Paris C Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Franc esca Arcelli Fontana, Terese Besker, Alexandros Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Nasia Moschou, **Ilaria Pigazzini**, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, Angeliki Agathi Tsintzira, "An overview and comparison of technical debt measurement tools", *IEEE Software*, vol. 38, no. 3, pp. 61-71, May-June 2021. Contribution: collected the data and wrote a small part of the paper.

Ilaria Pigazzini, Davide Foppiani and Francesca Arcelli Fontana, "Two different facets of architectural smells criticality: an empirical study" in Proc. Of the 1st International Workshop on Mining Software Repositories for Software Architecture, ECSA 2021 Companion Volume, 3 – 17 September, 2021, Virtual. Contribution: wrote the paper. [*Discussed*, Section 5.3]

Ilaria Pigazzini, Daniela Briola and Francesca Arcelli Fontana, "Architectural Technical Debt of Multiagent Systems Development Platforms", *in Proc. Of the 22nd Workshop From Objects to Agents (WOA)*, September 1-3, 2021, Bologna, Italy. Contribution: conceived and designed the analysis, collected the data, performed the analysis and wrote the paper. [*Discussed*, Section 5.2.3]

Rafael Capilla, Tommi Mikkonen, Carlos Carrillo, Francesca Arcelli Fontana, **Ilaria Pigazzini**, Valentina Lenarduzzi, "Impact of Opportunistic Reuse Practices to Technical Debt" , *in Proc. Of 2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 16-25, June 3, 2021, Virtual. Contribution: performed part of the analysis and wrote part of the paper. [*Discussed*, Section 5.2.1]

Ilaria Pigazzini, Francesca Arcelli Fontana, Bartosz Walter, "A Study on Architectural Smells and Design Patterns Correlation", *the Journal of Systems and Software (JSS)*, vol. 178, August 2021. Contribution: designed the analysis, collected the data, performed the analysis and wrote the paper. [*Discussed*, Section 4.2]

Andrés Diaz Pace, Antonela Tommasel, **Ilaria Pigazzini**, Francesca Arcelli Fontana, "Sen4Smells: A Tool for Ranking Sensitive Smells for an Architecture Debt Index", *in Proc. Of IEEE IEEE Congreso Bienal de Argentina (ARGENCON)*, pp. 1-7, 2020, Virtual. Contribution: performed part of the analysis. [*Discussed*, Section 5.2.4]

Francesca Arcelli Fontana, Federico Locatelli, **Ilaria Pigazzini**, Paolo Mereghetti, "An architectural smell evaluation in an industrial context", *in Proc. of International Conference on Software Engineering Advances (ICSEA)*, pp. 79-85, 8-22 October, 2020, Porto, Portugal. Contribution: wrote the paper. [*Discussed*, Section 3.2]

Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, Davide Taibi, "Towards Microservices Smells Detection", *in Proc. Int. Conference on Technical Debt (TechDebt)*, pp. 1-7, 28-30 June 2020, Seoul, Republic of Korea. Contribution: developed analysis tool, designed the analysis, collected the data, performed the analysis and wrote the paper. [*Discussed*, Section 6.2.1]

Francesca Arcelli Fontana, **Ilaria Pigazzini**, Claudia Raibulet, Stefano Basciano and Riccardo Roveda, "The PageRank and Criticality of Architectural Smells", The 6th Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon), *in Proc. 13th European Conference on Software Architecture (ECSA)*, pp. 197-204, 9-13 September 2019, Paris, France. Contribution: wrote the paper.

Ilaria Pigazzini, "Automatic Detection of Architecture Erosion through Semantic Representation of code", *in Proc. 13th European Conference on Software Architecture (ECSA)*, pp. 59-62, 9-13 September 2019, Paris, France. Contribution: conceived and designed the analysis, collected the data, performed the analysis and wrote the paper. [*Discussed*, Section 2.4]

Ilaria Pigazzini, Francesca Arcelli Fontana, Andrea Maggioni, "Tool support for the migration to microservice architecture:

an industrial case study", *in Proc. 13th European Conference on Software Architecture (ECSA)*, pp. 247-263, 9-13 September 2019, Paris, France. Contribution: conceived and designed the analysis and wrote the paper. [*Discussed*, Section 6.1.2]

Francesca Arcelli Fontana, Paris Avgeriou, **Ilaria Pigazzini**, Riccardo Roveda, "A Study on Architectural Smells Prediction", *in Proc. The 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 333-337, 28-30 August, 2019, Kallithea-Chalkidiki, Greece. Contribution: wrote part of the paper.

1.1.2 *Submitted papers*

Ilaria Pigazzini and Francesca Arcelli Fontana "Architectural Smells Evolution and Correlation: an Empirical Study", *submitted to the Journal of Systems and Software (JSS)*, 2021. Contribution: designed the analysis, collected the data, performed the analysis and wrote the paper. [*Discussed*, Section 4.3]

1.1.3 *To be submitted papers in November 2021*

Ilaria Pigazzini, Marco Belotti, Francesca Arcelli Fontana and Dario di Nucci, "Exploiting dynamic analysis for architectural smell detection: a preliminary study", *to be submitted to the Journal of Systems and Software (JSS)*, in November 2021. Contribution: performed the analysis and wrote the paper.

1.1.4 *Published papers not strictly related to the thesis*

Francesca Arcelli Fontana, **Ilaria Pigazzini**, Claudia Raibulet, "Teaching Software Engineering Tools to Undergraduate Students", *in Proc. 11th International Conference on Education Technology and Computers (ICETC)*, pp. 262-267, 28-31 October 2019, Amsterdam, The Netherlands. Contribution: collected data and wrote part of the paper.

2 ARCAN: A TOOL FOR ARCHITECTURAL SMELL DETECTION

This chapter introduces the tool for architectural smell detection, Arcan, which has been the base for developing most of the studies presented in this thesis.

Arcan was developed by us in the ESSeRE Lab of the University of Milano - Bicocca. The main aim of the tool is to detect architectural smells in Java projects, however during the past few years several extensions were applied to it. At the moment Arcan can run also on projects written in C/C++, it offers support for the migration towards microservices architecture (see Section 6.1.2) and it is able to compute an architectural debt index (see Chapter 5).

The main strength of Arcan is the dependency graph, a data structure which stores all the information related to the software architecture under analysis, starting from the representation of software components as nodes and their dependencies as edges. The graph allowed us to represent heterogeneous dimensions related to software architectures, e.g., static metrics, semantic information and structural information. During the various Arcan extensions, also the graph has been improved and complemented with other data structures (see Section 2.1).

The following sections describe Arcan architectural components and enabling technologies, as long as the definitions of the AS detected by the tool, with a brief discussion of their impact on software quality. We report also the detection strategies implemented for the identification of AS. We do not describe the details about the architecture and the Arcan enabling technologies, but they can be consulted in the main Arcan publications [19][21].

2.1 ARCAN COMPONENTS

Arcan relies on three main data structures: two graphs for the representation of the project under analysis and the Containment Tree, for the representation of the package structure. Most of the Arcan detectors are based on graph algorithms and metrics to evaluate specific properties of the project under analysis. Setting a threshold for each metric is not a trivial task, thus we implemented in Arcan a method [18] which allows to compute adaptive thresholds. We now briefly describe these main Arcan components.

THE DEPENDENCY GRAPH. It is the representation of the project under analysis in the form of a directed graph. The basic nodes represent the system components, such as Java classes, packages and methods. Edges represent the relationships among the various components and are divided in different types depending on the nodes they connect, for instance *use* relationship and *hierarchy* relationship. The benefits of exploiting a graph data structure are mainly two: first, for the detection of some kinds of smells, which can be identified through graph algorithms [19]; second, this structure allows to store all project information in a NoSQL schema-free graph database [167], enabling the reuse of the database and the flexibility of changing its schema basing on the detection needs. In our case, we exploit the Neo4j [173] graph database.

THE FEATURE GRAPH. It is the evolution of the dependency graph which enables the modelling of software concerns. In this research context, a *concern* is a software system's role, responsibility, concept, or purpose [251]. We propose this extension to enable the detection of two smells which violate the *Separation of concerns* principle, Feature Concentration and Scattered Functionality, described in Table 2.1. The feature graph associates to a set of project files (Java classes) a name in natural language (*feature*, as synonym of concern), enabling developers to *read* how features are disposed across the project. To identify which word best defines a feature, we exploit the tf-idf score [204], a well-known information retrieval metric: it reflects how important a word is to a document in a collection of documents. In the context of this work, a document corresponds to the set of words in natural language of a single Java class and a corpus is a collection of classes. The term which best indicates the feature for a given set of classes is the one with the highest tf-idf score. At the end of the generation of the features, the classes of a project are grouped by feature and, as consequence, packages are associated to a set of features. The Tf - Idf is defined as:

$$\mathrm{Tf} - \mathrm{Idf} = \mathrm{Tf}(t, d) \times \mathrm{Idf}(t, D) \tag{1}$$

where $\mathrm{Tf}(t, d)$ is the number of times that term t occurs in document d and $\mathrm{Idf}(t, D)$ is the logarithmically scaled inverse fraction of the documents d_i belonging to the corpus D that contains the term t.

We now describe in details the steps to be followed in order to generate the feature graph:

1. For each package, we identify a set of classes which share at least a use, hierarchy or interface dependency and we aggregate the text (the code) of the classes belonging to the same set.

2. We prepare the text in order to run the tf-idf score training. To do so, we apply some preprocessing techniques. *a)* We tokenize

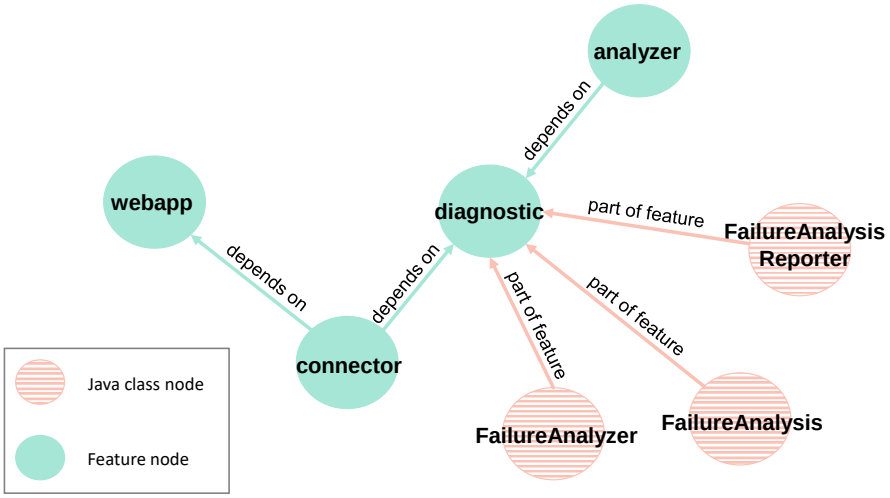


Figure 2.1: Spring Boot feature graph

the text, via a specific regular expression, since Java code follows the Java naming convention "Camel Case"[1] *b)* We normalize the tokens, by converting them to lower case and by removing numbers, punctuation and stop words, which are the very common words in a language. The stop words list for code comments depends on the used natural language (e.g. "the" token in English language) *c)* We apply stemming, that is the extraction of the morphological root of a word (e.g., "Play" is the root of "Playing").

3. We run tf-idf for each set of classes and then, still for each set, we choose the highest scoring word. Such word will indicate the feature associated to the set of classes.

As result, we obtain a new representation of the project. Classes are grouped by feature and, as consequence, packages are associated to a set of features.

Figure 2.1 illustrate an extract of the feature graph of the Spring Boot project. The striped, orange nodes represent Java classes which all belong to the feature node named "diagnostic". Notice that the feature nodes are linked one to another with the edge *depends on*: an edge appears between two features F1 and F2 if there is at least a dependency (either use or inheritance) between classes belonging to F1 and classes belonging to F2.

THE CONTAINMENT TREE. It represents the structure of the packages of a given project under the form of a tree where the root node is the root folder of the project and every node of the tree corresponds

1 CamelCase is the practice of writing compound words or phrases joining each word without spaces and capitalizing within the compound e.g. *startTopicDetection*

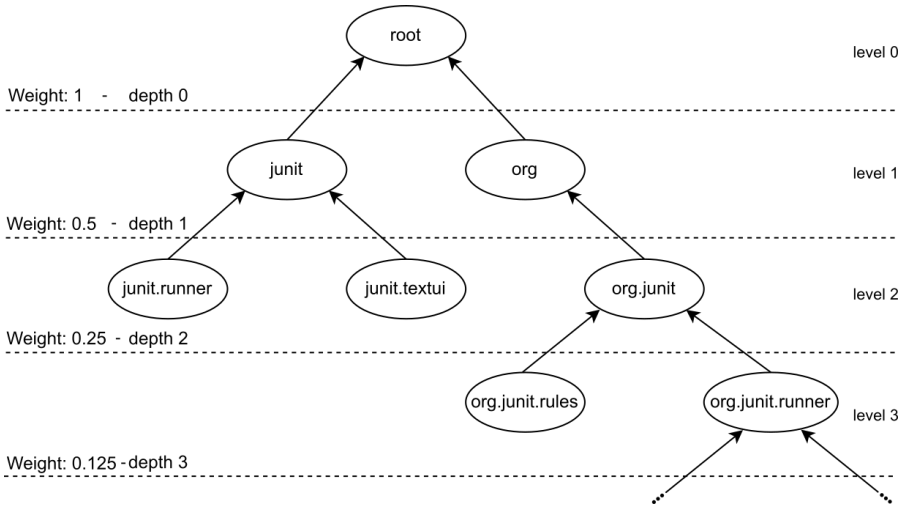


Figure 2.2: JUnit4 containment tree

to a project package. In Figure 2.2, a subtree of the entire containment tree of JUnit 4 [15] project is represented. Looking at this figure, it is possible to see the structure of the containment tree: at the higher level, *level 0*, we find the *root* node which is the parent node of the initial packages positioned in *level 1*. From *level 2*, there are all the sub-packages derived from the level 1 packages. All nodes are linked with their parents with an edge having as source node the child and as target node the parent. We exploit the containment tree for the detection of Scattered Functionality smell (see Section 2.3.6) and the computation of the *Diameter* metric, used to evaluate the cost-solving of Cyclic Dependency smell (see Section 2.3.3).

ADAPTIVE THRESHOLD GENERATION. Some of our detectors are based on metric thresholds. However, it is not trivial to define a static threshold able to fit every project under analysis. Hence, we adopted an adaptive method [18] able to generate a custom threshold given a metric of interest. The proposed approach respects the statistical properties of the metric, such as metric scale and distribution, furthermore it is repeatable, transparent and straightforward to carry out. We applied our approach for a set of AS whose detection is based on crossing a given metric(s) threshold. Given a project under analysis, we collect all the values of the metric to obtain a dataset, and then we generate the frequency distribution of 100 percentiles. The approach selects the median value of the distribution and searches for the percentile for which all values in higher position are lower or equal to the median. By applying this method, the most repetitive values are filtered out of the data set, and only the more variable values remain. The selected percentile is used as a cut-off point for the original data set and used to calculate the corresponding threshold value. In this

way, we can easily obtain a valid solution to the problem of setting a threshold value for quality metrics.

2.2 ARCHITECTURAL SMELLS DETECTED BY ARCAN

Currently, Arcan is able to detect six AS. Table 2.1 indicates for each detected AS its *Name*, its *Aliases* (if exist), its *Definition* and its *Consequences*, i.e., the shortcomings caused by the presence of the AS.

In the remaining of this section, we explain why we decided to focus on this specific set of smells and briefly introduce the concept of *criticality* and *cost-solving* of an architectural smell.

2.2.1 *Why did we decided to detect these smells?*

We decided to work on the smells of Table 2.1 for different features/peculiarities of each smell described below:

- **Cyclic Dependency (CD), Unstable Dependency (UD), Hub-Like Dependency (HL)** smells are based on dependency issues. Dependencies are of great importance in software architecture: components (class or packages) that are highly coupled and with a high number of dependencies are considered more critical, since they have higher maintenance costs. In particular, in the opinion of some developers, Cyclic Dependency is one of the most common and most critical smell [155] and according to another empirical study UD smell is one of the most common AS too [216]. The description of Arcan detection strategy for these three smells can be found in Section 2.3.

- **God Component (GC)** is a problem about the size of the components, not strictly related to the dependency structure of the system but to how the system is *modularized*. Indeed, GC violates the modularity principle [151]. At the moment, the Arcan detection strategy consists in simply computing the number of Lines Of Code (LOC) belonging to each system package. In the future, we could enhance the detection with a more sophisticated strategy.

- **Feature Concentration (FC) and Scattered Functionality (SF)** regard how software concerns are implemented in the software architecture: a good design should follow the separation of concerns principle [251], where each architectural component addresses a separate concern. When concerns are well-separated, individual parts of the architecture can be reused, as well as developed and updated independently [144]. Arcan detects FC by checking for each package its associated features, thanks to the

Table 2.1: Architectural smells definitions

Name	Aliases	Definition	Consequences
Cyclic Dependency	Dependency Cycle, Tangle, Cross-Module Cycle, Cross-Package Cycle, Cycle of classes, Cyclically-dependent Modularization	Refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem's dependency structure.[148]	The components involved in a dependency cycle can be hardly released, maintained or reused in isolation. Moreover, a change on one affected component will propagate towards all the other ones involved in the cycle.
Feature Concentration	Concern Overload	This smell occurs when an architectural entity implements different functionalities in a single design construct. [14]	This smell violates the separation of concerns and the single responsibility principles, moreover the components it affects are hard to understand and maintain.
God Component	God Class, Blob	This smell occurs when a component is excessively large either in terms of LOC (Lines Of Code) or number of classes. [144]	God Components are hard to understand, hard to modify, hard to reuse and cause changes ripple effect. Moreover, they are hard to test because they usually comprise too many functionalities.
Hub-Like Dependency	Link Overload, Crossing, Hub Like Modularization	This smell arises when a component has (outgoing and ingoing) dependencies with a large number of other abstractions. [**Suryanarayana2015**]	The component in the middle of the hub is a unique point of failure and a dependency bottleneck. Moreover the logic inside a Hub-Like Dependency is hard to understand, and the smell causes change ripple effect.
Scattered Functionality	Scattered Parasitic Functionality	Describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. [90]	This smell violates the separation of concerns principle, it is hard to maintain and it is hard to understand how and where the functionality is implemented in the system.
Unstable Dependency	Unstable Interface	Describes a subsystem (component) that depends on other subsystems that are less stable than itself. [148]	The components with an high instability are more prone to change with respect to the more stable ones, this means that the component which depends on less stable components is forced to change along with them.

feature graph (see Section 2.1). If the package is associated to *too many* different features, then it is affected by FC. While concerning SF detection, Arcan checks for each feature if the feature spreads across packages belonging to different branches of the containment tree, and in that case it marks it as affected.

The definition of God Component and Feature Concentration is similar, however they represent two different problems: a GC is primarily a very large (in terms of LOC) component. FC emerges when there is a lack of cohesion and the component implements too many different purposes. The two smells can co-affect the same component (see Section 4.3). In that case, the component is large in terms of LOC and addresses too many features at the same time. In any case, Arcan exploits two different approaches to detect them, thus we study the two smells separately.

2.2.2 *Architectural smells criticality and cost-solving*

As for code smells [215], also for architectural smells it is important to evaluate the criticality of the smells, in order to prioritize the smells to be removed first. In such terms, *criticality* of an AS models the degree of removal urgency associated to the AS. However, it is not trivial to model and evaluate the importance and urgency of the removal of an AS. In the literature, the identification of the best metrics to be used for the evaluation of criticality is considered a complex task [249], mainly because it is tightly connected to how smells are perceived by developers [235] and such perception is subjected to many variables, such as the developer experience, code ownership [190], whether the smell is located in a central part of the project and other facets.

On the other hand, *cost-solving* (cost of fixing, cost of refactoring) of AS is the effort needed to remove a smell from the system [209]. This variable depends less from the perception of the developers but more from the specific characteristics of the interested AS.

Both criticality and cost-solving are particularly relevant for developers when making decisions about AS management: for instance, to choose which smell to refactor first [155][190]. A developer may prefer to refactor first the smells which require less time to be solved to quickly enhance the quality level of the project, instead of fixing the most critical ones. On the other hand, the developer may decide to remove the most difficult/critical ones, but to make this decision, different factors must be considered: it can be too expensive and risky; too many changes could compromise other parts. Perhaps, the most difficult AS was created by design choice and no better solution is available, as in the case of cycles created by callbacks for event listeners in GUI components [155][199]. Finally, the most critical AS

could appear in a not-central part of the project, such as a deprecated, unessential package, and could be not interesting for the developers.

We defined a set of metrics to evaluate the cost-solving of a given architectural smell instance and they are described in Section 2.3. Concerning criticality, we measure it with PageRank, a measure inspired by the well-known metric from Brin and Page [41] that estimates whether an AS is located in an important part of the project [271], where the importance is evaluated according to how many parts of the project depend on the ones involved in the AS (as a sort of centrality measure of the AS). We use PageRank as a proxy of AS criticality, i.e., the higher the PageRank, the higher the criticality of the AS.

We conducted a study on architectural smells criticality and cost-solving (see Section 5.3). Moreover, we empirically evaluated the perceived criticality of the different types of smells in two industrial studies (see Section 3).

2.3 ARCAN DETECTION STRATEGIES

The following section describes the detection strategies implemented in Arcan for the identification of the architectural smells reported in Table 2.1.

Most of the strategies exploit the dependency graph by running graph algorithms to understand if a certain structure (e.g., a cycle) is present in the graph. Some check whether the architecture under analysis overcomes the threshold value of specific also the computation of static metrics (e.g., number of Lines of Code) and finally for the detection of two smells, Scattered Functionality and Feature Concentration, the strategies relies on Natural Language Processing (NLP) metrics.

For each type of smell we describe:

- **Granularity level**: the type of architectural component on which the smell can be detected (Java class or package).
- **Alternative Names**: aliases of the smell that can be found in the literature.
- **Implementation**: the description of the detection strategy implemented in Arcan.
- **Cost-solving evaluation**: the metrics and techniques used to evaluate the cost-solving of the AS instances. We also use the term *Severity* to indicate a generic metric used for cost-solving estimation. Cost-solving values can be used to discriminate the different smell instances by ordering the AS from the most difficult to refactor to the less difficult. The cost-solving of an AS is the evaluation of the impact on the removal of the AS of specific AS's features. This is an important aspect to consider when

dealing with smells and their refactoring because knowing how much a smell is difficult to remove can be determinant during the refactoring phase, when developers must choose which smell to prioritise. We propose a set of metrics to automatically evaluate the cost-solving of AS, based on the characteristics of the different types of smells. There are two kinds of metrics: the ones which are also used during the smell detection and whose thresholds allow to establish whether the smell is present or not, and the others which measure specific smell properties, not used during the detection, but used for computing cost-solving.

- **Also detected by**: the names of the other tools that can detect the smells.

2.3.1 *Unstable Dependency (UD)*

Granularity level: detected on packages.

Alternative Names: There are no known alternative names (according to our knowledge).

Implementation:

- Input: a subgraph of the original dependency graph, where the only nodes are packages and the edges represent the afference between packages, i.e., the dependencies among packages.
- Exec: the detector computes the Instability metric [150] for every package. For every package, the detector checks if it is afferent to a less stable package; if so, put it in a map with the list of related less stable packages.
- Output: a map with every package affected by the smell and the associated packages which caused it.

Cost-solving evaluation:
Each of the following metrics allows to discriminate UD smells by ordering the most difficult to remove ones from the less difficult. They regard different aspects of the smell and all of them can be used to obtain a smell ranking.

- **Instability (I):** The ratio of outgoing dependencies on the total number of dependencies of a package [150]. This metric evaluates the package's resilience to change and the UD detection is based upon it. It ranges in [0,1], where 0 indicates that the package is completely stable and 1 completely unstable.
- **Degree of Unstable Dependency (DoUD):** The ratio between the number of dependencies that point to less stable packages and the total number of dependencies of the package. Its range

is in $(0, 1]$. The higher the metric, the higher the chance a change occurs and propagates to the affected components, because of its multiple less stable dependencies.

- **Instability Gap (IG):** The difference between the instability of the affected package and the average instability of the dependencies less stable of the package itself. Its range is in $(0, 1]$. The higher the instability gap, the higher the chance the package affected by the smell is changed due to ripple effects.

The worst case scenario occurs when all package dependencies are unstable and with the maximum instability gap. In that case, the criticality is the highest possible. Moreover, also the Instability of the packages involved in the smells is a metric of interest, since higher instability means a higher change proneness, thus a higher cost-solving.

Also detected by: Designite and Dv8.

2.3.2 *Hub-Like Dependency (HL)*

Granularity level: detected on classes and packages.

Alternative Names: Hub Like Modularization, Link Overload.

Implementation: We now report the implementation of the Hub-Like Dependency on *classes*. The procedure is the same for packages.

- Input: a subgraph of the original dependency graph, where the only nodes are classes and the edges represent the dependencies among classes.
- Exec: for all class nodes, the detector computes the ingoing and outgoing dependencies; then, the detector calculates the median of the number of ingoing and outgoing dependencies of all the classes of the system; the detector checks if the number of ingoing and outgoing dependencies of a class is respectively greater than the ingoing median and outgoing one; then, the detector checks if the difference between ingoing and outgoing dependencies is less than a quarter of the total number of dependencies of the class; finally, in order to keep only the classes with an exceptional high number of dependencies, we compute the adaptive threshold (see Section 2.1) of the Total Number of Dependencies metric. If the class's dependencies are over the threshold, then the class is considered a hub.
- Output: a map of the classes affected by the smell and their relative Fan In and Fan Out metric values [150].

Cost-solving evaluation:
We consider four metrics to evaluate HL criticalities:

- **FanIn and FanOut:** indicate respectively the ingoing and outgoing dependencies [150].
- **Total Dependencies**: the total number of dependencies of a software component (method/class/package). It corresponds to the sum of the number of ingoing dependencies (from other classes/packages into the affected one) and the number of outgoing dependencies (vice-versa) i.e. the sum of FanIn and FanOut.

Fan In, Fan Out and Total Dependencies gives information about the size of the smell i.e. how many dependencies are affected by the smell. The higher their values, the higher the smell cost-solving.

Also detected by: AI Reviewer, Arcade and Designite.

2.3.3 *Cyclic Dependency (CD)*

Alternative names: Tangle, Cross-Module Cycle, Cross-Package Cycle, Cycle of classes, Cyclically-dependent Modularization.

Granularity level: detected on classes and packages.

Implementation:

- Input: a subgraph of the original dependency graph, where the only nodes are classes or packages depending on the requested granularity level.
- Exec: the detector launches the Depht First Search (DFS) algorithm on the subgraph and collects every node involved in a cycle in a different list.
- Output: a list for every cycle detected by the DFS algorithm.

Cost-solving evaluation: We consider the following 4 metrics for the evaluation of Cyclic Dependency cost-solving:

- **Severity:** describes the *structural composition* of an AS i.e. the classes/packages/methods involved in the smells and the dependencies which form the smell. For CD at package level is defined as follows:

  $$Severity = 1 - (1/NumC + NumP \times NumP/NumC)$$

  where NumP is the number of packages involved into the cycle and NumC is the number of classes contained into the affected packages, whose inter-packages dependencies cause the creation of the package cycle. For smells at class level, the formula becomes:

  $$Severity = 1 - (1/NumM + NumC \times NumC/NumM)$$

where NumC is the number of classes involved into the cycle and NumM is the number of methods contained into the affected classes whose inter-classes dependencies cause the creation of the package cycle.

- **Diameter:** computed only for CD at package level. It indicates the **worst possible distance between two packages contained in a cycle**[131] and is based on the assumption that very distant packages implement different system functionalities. The Diameter metric is defined as:

  $$Diameter(C) = max(\delta(x,y)|x,y \in P(C), x \neq y)$$

  where *C* is a cycle, *P(C)* is the set of packages contained in the cycle and δ is the distance between two packages in the containment tree (see Section 2.1). This distance is obtained computing the *Weight* metric, defined as:

  $$Weight = 1/2^d$$

  where d is the depth of the edge in the containment tree.

Severity metric assumes higher values when the cycle forms a complex structures in the dependency graph. A complex structure means that many components (class/packages) are involved and must be addressed during refactoring. Diameter metric can be used to evaluate the cost-solving, but only for cycles among packages. Cycles among distant packages (in different branches of the containment tree) mean that parts of the system which should be detached and should implement different concerns, are wrongly tighten by a circular dependency. Hence higher Diameter values correspond to higher cost-solving.

Also detected by: Arcade, AI Reviewer, Designite, Sonargraph, Dependency Finder, JArchitect, ClassCycle, and NDepend.

2.3.4 *God Component (GC)*

Alternative names: God Class, Blob.

Granularity level: detected on packages.

Implementation:

- Input: a subgraph of the original dependency graph, where the only nodes are packages.
- Exec: for every package, the detector computes the number of Lines Of Code belonging to the package. If this number is over a its adaptive threshold, then the package is affected by the smell.
- Output: a table with every package affected by the smell and the number of classes which cause the smell.

The Lines of Code adaptive metric is computed through the adaptive procedure (see Section 2.1).

Cost-solving evaluation: We consider the following metrics for the evaluation of God Component:

- **Lines of Code (LOC):** the number of lines of code of all classes contained in the package.
- **Lack of Component Cohesion (LCC):** to measure the internal cohesion of the package affected by the GC smell.

A very large package with low cohesion is more critical because it means that the package holds too many responsibilities inside the project; on the other hand, a very large package with a high cohesion is more difficult to refactor (in terms of dividing it in smaller packages).

Also detected by: AI Reviewer, Arcade and Designite.

2.3.5 *Feature Concentration (FC)*

Alternative Names: Concern Overload. [133]

Granularity level: detected on packages.

Implementation:

- Input: a subgraph of the original feature graph, where the only nodes are packages and the feature nodes (see Section 2.1).
- Exec: for each package, the detector collects its associated features from the feature graph. If the package is associated to *too many* different features, then it is affected by feature concentration.
- Output: a table indicating for each package the number of distinct features.

We quantify the expression *too many* through an adaptive threshold (see Section 2.1).

Feature Concentration is one of the new AS whose detection is introduced in Arcan for the first time. Our implementation differs from the one proposed by Garcia et al. [92] since we do not exploit a topic model, but the feature graph, based on the tf-idf metric (see Section 2.1). Our approach has the advantage that does not need an input (i.e., the number of topics needed by the LDA algorithm) and thus is completely automatic. It is also different from the one of Sharma et al. [221], since they only consider static dependencies to detect this smell, while we also exploit the additional information provided by the feature graph. In our opinion, the semantic information must be

considered when we aim to identify smells which impact on the separation of concerns principle. We try, by considering the words in natural language coming from the code, to reverse engineer the abstract concepts addressed by the architectural components.

Cost-solving evaluation:
The metrics used to compute the cost-solving of this smell is:

- **Number of Features (NoF)**: the number of distinct features associated to the affected package.

Packages with a higher number of distinct features imply longer time for their removal: in this case a smell with high NoF is more critical.

Also detected by: Arcade, Designite.

2.3.6 *Scattered Functionality (SF)*

Alternative Names: Scatter Parasitic Functionality [92].

Granularity level: detected on packages.

Implementation: In order to detect this smell, Arcan relies on the feature graph and on the containment tree (see Section 2.1). The feature graph is exploited to collect features (in this context, we assume that a functionality can be represented by the detected features), while the containment tree is exploited to locate features across system packages. As for Feature Concentration, we differentiate from the detection of Garcia [92] and Sharma [221] by exploiting the feature graph with the tf—idf score.

- Input: a subgraph of the original feature graph, where the only nodes are packages and the feature nodes.
- Exec: For each feature, if the feature spreads across packages belonging to different branches of the containment tree, than this is an instance of Scattered Functionality.
- Output: a table indicating for each features the packages which are associated to it.

Cost-solving evaluation: The metric used to compute the cost-solving of this smell is:

- **Scatter:** a measure of how much the feature is scattered over the project. It is measured as the number of packages in which the scattered feature is implemented.

Scatter metric reflects how many parts of the code should be taken in consideration during refactoring, hence the higher this metric is, the higher the smell cost-solving.

Also detected by: Designite.

2.4 AUTOMATIC DETECTION OF ARCHITECTURAL SMELLS THROUGH SEMANTIC REPRESENTATION OF CODE

As already explained, we implemented in Arcan a component to retrieve semantic information from code and store them in the *feature graph*. However, prior to the choice of exploiting the tf-idf metric to model code semantic, we explored the possibility of adopting a deep learning model to reach the same goal[2].

In particular, we tried *code2vec*, a neural model for representing snippets of code as countinuos distributed vectors (*code embeddings*) [12]. The code embeddings approach allows to associate a continuous distributed vector to a piece of code and compute the semantic similarity between different pieces of code. This allows to have semantic information bounded to specific parts of code and to perform different tasks. For instance, to understand how a specific concern of the system spreads through code and consequently identify anomalies such as architectural smells. We directed this study by posing a research questions:

- *RQ: "Is it possible to represent software concerns with the code2vec model?"* In order to answer the question, we investigated whether the distributed code vector space is able to represent the semantic properties of the software architecture. Code2vec takes as input generic snippets of code. However, in order to simplify the problem and exploit the original formulation of the model and its provided implementation, we started from the investigation of how Code2vec represents *methods* in Object-Oriented (OO) code.

We implemented the study by developing a small Python application able to query the code2vec model and collect sematic information for all the method dependencies of a given Java project.

2.4.1 *Description of the approach*

The aim of the study was to investigate how concerns spread through the architecture and to understand whether architectural smells can be detected by leveraging code embeddings representation. We now describe our approach and propose two detection strategies for Scattered Functionality and Feature Concentration, the two AS which violate the separation of concerns principle (see Section 2.2).

Both the two detection strategies leverage the graph representation of the structural dependencies in the project under analysis (*dependency graph*) combined with the semantic information extracted by the code2vec model. In particular, we exploit the ability of the model

2 A publication was extracted from this study [195]

to generate vectors representing the semantics of the input methods. Hence, given an Open-Source project, the architecture of the project is represented as $G(V, E)$, where V is the set of nodes comprehending methods, classes and packages of the project and E is the set of dependencies among them. Moreover, given the set of methods $M \subset V$, the code2vec model creates a vector space where each point corresponds to a method.

2.4.2 *Analysis of the vector representation*

The first part of the study focused on understanding whether code embeddings are suitable for the representation of concerns inside software architecture. In particular, we were interested in investigating if vector similarity is a proper metric to quantify the semantic dependency among different methods in Object-Oriented projects. The vector similarity considered in this study is the *cosine similarity*, which measures the cosine of the angle θ between two non-zero vectors $\mathbf{v}$ and $\mathbf{w}$.

$$similarity(\mathbf{v}, \mathbf{w}) = \cos\theta = \frac{\mathbf{v} \cdot \mathbf{w}}{|\,\mathbf{v}\,||\,\mathbf{w}\,|} \tag{2}$$

This metric is non-negative and bounded between $[0, 1]$. We chose this metric since it is commonly used in NLP to measure document/text distance.

The analysis consisted of the following steps:

- Generation of the code vectors associated to the methods of an Object-Oriented project with code2vec.
- Computation of the vector similarity of all methods of the OO project. The Cosine Similarity metric is computed for each pair of vectors without taking into account the order, hence the total number of combinations is
$$\sum_{i=0}^{n-1} n - i = \frac{n(n+1)}{2} \tag{3}$$
where n is the number of vectors.
- Compare the similarity distribution of *virtual edges* and *concrete edges*. For virtual edge we mean the possible edge that can link two methods; since similarity is computed for every couple of methods, there is a similarity value for each virtual edge. Instead, a concrete edge is an actual dependency in the project.

The aim of the proposed analysis was to understand if similarity follows a particular distribution and to investigate if the distribution changes when considering couples of methods which actually are linked one to another. If proved on a meaningful number of Object-Oriented projects, a non-random similarity distribution could be the first sign of significance of the similarity metric.

2.4.3 *Architectural Smells detection*

The second part of the study aimed to propose the detection strategies for Scattered Functionality and Feature Concentration smells based on the code2vec model exploitation.

SCATTERED FUNCTIONALITY The detection of Scattered Functionality exploits the similarity metric to build paths on the dependency graph which represent software concerns. The paths consist of a set of edges, e_t^*, which have the property of maximizing the similarity value between code vector m_z and the sum of the two antecedent code vectors m_x and m_j. The formal definition of *concern path* is the following:

Let $M \subset V$ be the set of methods m_i of the project. Let $E_m \subset E$ be the set of method dependencies. Let $G(M, E_m)$ the induced directed graph of $G(V, E)$ having for nodes M and for edges E_m. Let $M_{src} \subset M$ be the set of source vertices of the graph. Let $M_{snk} \subset M$ be the set of sink vertices of the graph.

- For each $m_i \in M$, compute the associated code vector.
- For each m_{src}, for each $y_{src} \in neigh(m_{src})$

$$
\begin{aligned}
e_0^* &= m_{src} \to y_{src} \\
p^* = &\{e_t^* \mid e_t^* = y \to z, \\
&\quad {}_{m_z} similarity(m_x + m_y, m_z), \\
&\quad m_x = out(e_{t-1}^*), \\
&\quad m_y = in(e_{t-1}^*), \\
&\quad t \in \{1..n\}\}
\end{aligned}
\tag{4}
$$

where n is the depth of the first sink node encountered along the concern path. Hence, the strategy to detect Scattered Functionality on a OO project consists in the following steps:

- Compute all the concern paths of the dependency graph of project
- If a computed path crosses more than a package, the involved packages are affected by the smell.

FEATURE CONCENTRATION The detection strategy proposed for this smell exploits the semantic vector similarity to run a clustering algorithm. The aim of the clustering analysis is to group similar methods inside a given package: we hypothesized that the detected groups correspond to the concerns of the package. Then, if a package shows too many concerns, it may be affected by Feature Concentration smell. The proposed detection strategy consists in:

- For each package, for each couple of methods $m_i \in c_i, m_j \in m_j$, $c_i \neq c_j$, compute $similarity(m_i, m_j)$.
- Run a clustering algorithm to identify the groups of similar methods
- If the number of detected groups and the similarity distance among them is high, the package is affected by Feature Concentration.

The choice of a suitable clustering algorithm was not part of this (exploratory) study.

2.4.4 *Findings*

Figure 2.3 and 2.4 show the similarity distributions of project Checkstyle v5.6 [238], computed on the 1306 methods belonging to package *checks*. We chose Checkstyle because we are familiar with it (suitable for future manual validation) and it is a small-sized project, on which the model can be run even with limited resources. The similarity values shown in Figure 2.3 were computed on all the possible combinations of different methods (*virtual edges*), while the ones in Figure 2.3 were computed on the couples of methods actually linked (*concrete edges*). The former are bounded in $[-0.3620, 1]$, while the latter in $[-0.1994, 0.9873]$: this means that, for Checkstyle, the set of possible values which model concrete edges among methods is smaller with respect to the one which model all possible edges. Such finding encourages further analysis in order to understand the significance of similarity. Moreover, the virtual edges distribution shows anomalous spikes for certain similarity values, which may be caused by possible approximations of the algorithm used during the similarity computation. Concerning the concrete edges distribution, the plot shows a concentration of values around 0.1 and 0.25. We tried to fit it as a mixture of two normal density distributions using the Expectation Maximization algorithm and it resulted that the data could belong to two different distributions.

2.4.5 *Final remarks*

RQ: Is it possible to represent software concerns with the code2vec model? From our study, we found that code2vec has the potential to produce the representation of software concerns. In particular, cosine similarity computed on code2vec vectors seems to provide additional information (method cohesion) to be attached to software dependencies, i.e., information about how much two methods are involved in the same concern.

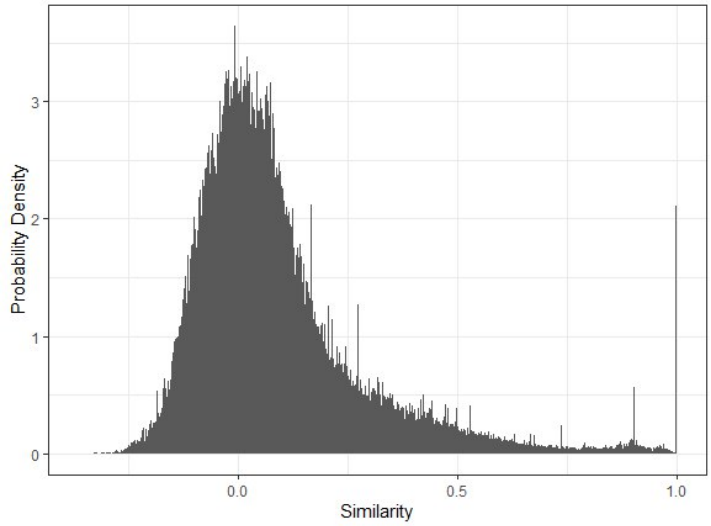


Figure 2.3: Checkstyle similarity - virtual edges

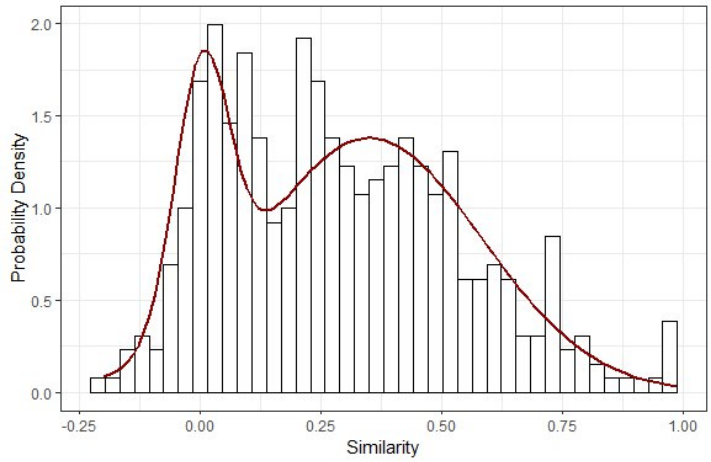


Figure 2.4: Checkstyle similarity - concrete edges

However, we chose not to implement such approach in Arcan due to the high computational costs required by the model to generate a vector for each method and required by our Python script to compute the cosine similarity between method dependencies.

In any case, the research about neural networks models is still open and an interesting future work could be the implementation of the proposed detection strategies for Scattered Functionality and Feature Concentration.

2.5 OTHER TOOLS FOR ARCHITECTURAL SMELLS DETECTION

We report in Table 2.2 a set of academic and commercial tools which detect the same architectural smells identified by Arcan. In general, the commercial tools offer the detection of only Cyclic Dependency, with the exception of AI Reviewer, which supports also the detection of God Component and Hub-Like Dependency, and DV8, which supports the detection of Hub-Like Dependency and Unstable Dependency.

On the other hand, the two indicated academic tools, Arcade and Designite, detect all of them. The difference between Arcan and the other two tools relies in the detection strategies. Table 2.3 briefly resumes the main differences between Arcan detection strategies with the ones of Arcade and Designite. For each architectural smell, we indicate the strategy adopted in Arcan (the rows highlighted in grey), the strategies implemented by either Arcade or Designite (*Detection strategy* column) and specify whether the approach is the *Same*, or uses a *Different threshold* (same approach, different threshold value for the metric to overcome) or consists of a complete *Different strategy* (*Comparison* column).

While for the detection of dependency issues such as CD and HL the approaches are equal or only differ for the value of some metric thresholds, concerning SF and FC (the smells violating the separation of concerns principle) the three tools adopt completely different approaches. In particular, the differences rely in how the the tools represent a software *concern* (a system's role, responsibility). Designite only exploits a static metric named Lack of Component Cohesion (see Section 2.2) to detect concerns, while Arcan and Arcade take both advantage of existing Natural Language Processing (NLP) techniques. Arcade models software concerns as *topics*, learned from source code with the Latent Dirichlet Allocation model, defining a topic as a probability distribution over the system's nonempty set of keywords, whose elements are used to describe the system (e.g., via comments in source code). By examining the words that have the highest probabilities in a topic, the meaning of that topic may be discerned and associated to a specific part of the system (usually a component such as a package). Arcan instead computes the Tf-Idf (see Section 2.1), a metric able to assign the highest score to the term (extracted from code) which better describes the concern associated to a specific piece of code. The strength of the Arcan approach it that is does not need additional information other than source code to run. The topic model approach instead requires the input of a set of hyperparameters (parameters whose value is used to control the learning process) and also the number of expected topics. Such inputs should be provided by the developer using the tool, making less actionable the tool itself.

Table 2.2: Academic and commercial tools for architectural smells detection

	Tool	CD	FC	GC	HL	SF	UD
acad.	Arcade [132]	x	x	x	x	x	
	Designite [64]	x	x	x	x	x	x
commercial	AI Reviewer [6]	x		x	x		
	DV8 [48]	x			x		x
	JArchitect [110]	x					
	Massey Architecture Explorer [158]	x					
	Ndepend [170]	x					
	Sonargraph [270]	x					
	STAN [180]	x					
	Structure101 [103]	x					

2.6 SUMMARY OF THE FINDINGS

We presented in this section our tool, Arcan, starting from the description of its components and the data structures it exploits. We also introduced the AS studied in this thesis and their detection strategies. We also reported the other academic and commercial tools which detect the same set of AS, highlighting the differences between their detection strategies and the Arcan approach.

We reserved particular attention to the description of our method to represent software concerns. We first explained our current approach, which is based on tf-idf and is an enriched version of the dependency graph (feature graph). Then, we reported our experience with a deep learning model for the semantic representation of code. Finally, we acknowledged that using neural models for such a purpose is promising but also resource-consuming, making them unsuitable, at the moment, for the integration with a detection tool such Arcan.

Table 2.3: Comparison with Arcan detection strategies

AS	Tool	Strategy
CD	Arcan	Depth-first search algorithm
	Arcade	Strongly connected subgraphs
	Designite	Depth-first search algorithm
FC	Arcan	#numberOfFeatures threshold overcome
	Arcade	topics threshold overcome
	Designite	Lack of Component Cohesion (LCC) threshold overcome
GC	Arcan	#Components/LOC threshold overcome
	Designite	#Components/LOC threshold overcome
HL	Arcan	Fan In and Fan Out threshold overcome
	Arcade	Fan In and Fan Out threshold overcome
	Designite	Fan In and Fan Out threshold overcome
SF	Arcan	Feature spreads across multiple components
	Arcade	Misplaced topics
	Designite	Number of methods accessing same component
UD	Arcan	Package instability comparison
	Designite	Package instability comparison

3

VALIDATION AND PERCEPTION OF THE ARCHITECTURAL SMELLS FROM THE DEVELOPERS

An important and challenging task when developing software analysis tools is their validation. Arcan suffers from false positives and in the past few years we conducted a set of validation studies, primarily in industrial context, to quantify the analysis precision. In all these studies, validation is intended in terms of discerning Arcan true positives from smells' instances correctly detected by Arcan but, for some reasons, not considered problematic by developers. For instance, because the code was designed in that way *on purpose* and thus it does not represent a problem to be solved. We call them false positive instances, even if the structure of the smell is actually present in the code. Table 3.1 describes how to interpret the architectural smells confusion matrix. For *Actual AS* we mean the instances of AS which are present in the code and at the same time are perceived as a problem by developers. For *Detected AS* we mean the instances of AS detected by Arcan.

Table 3.1: Architectural smells detection confusion matrix

		Actual AS	
		TRUE	**FALSE**
Detected AS	**TRUE**	**True positives:** AS detected which are real problems	**False positives:** AS detected which are not real problems
	FALSE	**False negatives:** AS not detected which are real problems	**True negatives:** AS not detected which are not real problems

In our studies, we compute the *Precision* of Arcan results, and when possible *Recall*. Precision is defined as the *fraction of actual AS* (True Positives, TP) among the *detected AS* (the sum of TP and False Positives, FP), thus indicating the ability of the tool of correctly identifying AS:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5}$$

Recall is defined as the *fraction of actual AS* that were detected by Arcan, thus quantifying how many TP the tool can detect over the *total number of actual AS* (the sum of TP and False Negatives, FN):

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

Recall is especially difficult to compute for us, because it requires a previous detailed knowledge of the project under analysis, i.e., the true number of AS present in the project. In our studies, we mainly evaluate precision, since developers hardly know about the AS of their projects.

Following in this chapter, we list the past validation studies conducted on the Arcan tool results and discuss in detail two recent studies on the perception of practitioners about the AS detected by Arcan.

3.1 PAST STUDIES ON THE VALIDATION OF ARCAN TOOL

The Arcan detection results of CD, UD and HL smells have been validated in our previous works. In particular, a validation of Arcan results has been performed on ten open source projects [19] and also in industrial contexts through the developers feedback on the detected AS instances. In a first study, the practitioners feedback about two projects assigned a high precision value of 100% to the results and 63% of recall [21]; in another study [155], practitioners gave feedback also about how they perceive AS and their possible refactoring, through the analysis of four industrial projects. In this case the overall precision was 50%. Developers provided insights about why some instances were not real problems for them, and in the case of CD they reported the case of cycles created by callbacks from anonymous classes in GUI components. Developers explained that callbacks for event listeners in the GUI components could not be easily replaced, and therefore they did not recognize CD as problematic in those specific cases, but rather as a *necessary solution*. In the case of UD, developers admitted that they did not easily understood its meaning and this could be a cause behind their disagreement on it being a problem. However, UD was also sometimes related to the use of the Strategy design pattern. The relationship between AS and DP is not unexpected and we further studied it in another work (see Section 4.2), where we performed a quality analysis on some examples of AS and DP collocation and found that there are some connections between the co-occurrence of specific types of AS and DP. Some of them are effects of AS false positives instances; others indicate that specific implementations of DP can imply the introduction of AS, as it happens for the Visitor pattern, which can cause the introduction of CD. Concerning the smell's impact on quality, in their opinion, HL is the one which impacts the most the system quality and also the

smell which gets worse the most during the project evolution. They also indicated CD as one of the most impactful smell, and also as the smell which requires more time and creates more side effects when refactored, even if in this study many CD instances were indicated as false positives, i.e., true problems but not to be removed.

3.2 AN ARCHITECTURAL SMELL EVALUATION IN AN INDUSTRIAL CONTEXT

According to the other AS, we conducted another validation study[1], always in an industrial context, where in addition to the above three smells, we validated GC, FC and SF. Also in this case we collected data about how the practitioners perceive AS. In particular, a survey with different questions was given to the practitioners. They were three and they were all developers belonging to the team that was working on the analyzed project at the time the survey was proposed. The first one was a junior developer with 4 years of experience working on the project analysed in this study. The second one was a middle developer with 9 and a half years of experience of which 1 year and a half spent working on the project. The third one, the team leader, was a senior developer with almost 15 years of experience working on the project for 2 years.

We presented 19 AS instances, drawn from all the detected smells, because we tried to include instances with different granularity (for CD and HL) and different metric values (for FC and GC).

The survey contained 12 questions that the three practitioners had to answer individually for each selected AS instance. We investigated three main aspects: *1)* Arcan detection precision; *2)* architectural smells perception and impact; *3)* architectural smell refactoring and criticality.

FEEDBACK ABOUT ARCAN DETECTION PRECISION The overall precision of the tool was 70%. Developers provided some explanation about the false positives instances: one HL was detected on a package containing utility classes which was supposed to be used by many other classes on purpose, thus making it a design choice; the size of one GC was justified by the attempt of developers of avoiding boilerplate code[2], still making it a design choice; finally developers indicated all SF instances but one as false positives because their design was layered (one layer per package) and not organized by feature (one feature per package). this kind of smell is meant to point out defects in a package-by-feature [134] organization which is desirable in some cases, but not when the actual design is layered, as the project

1 A publication was extracted from this study [79], in collaboration with Federico Locatelli and Paolo Mereghetti

2 Sections of code that have to be included in many places with little or no alteration.

analyzed in this study, where the smell was detected each time a vertical feature of the architecture was scattered among the packages. Developers got aware of that and signaled it to us, except for the case they considered true.

FEEDBACK ABOUT AS PERCEPTION AND IMPACT Concerning the practitioners' perception of AS, all the smell types were considered affecting maintainability by at least one developer. For one GC instance a developer suggested an additional aspect, that "they affect the domain structure" i.e. how the domain model is organized across the different packages.

We also investigated which AS type, in the opinion of the developers, gets worse as time passes, when left in the system. We discovered that the developers found Hub-Like on classes and Feature Concentration the most problematic in these terms.

FEEDBACK ABOUT REFACTORING AND CRITICALITY Concerning refactoring, developers pointed out that the refactoring of the ones regarding the separation of concerns (FC and SF) is crucial when migrating from a monolithic architecture to microservices. This because these types of smells affect how the system functionalities are organized, and a microservices migration requires to identify, isolate and put in the same microservice all the classes that work on the same functionality.

Moreover, they confirmed some of the results of the previous study, indicating HL as the most critical smell (see Section 2.2.2).

3.3 THE PERCEPTION OF ARCHITECTURAL SMELLS IN THREE SOFTWARE COMPANIES

In addition to the previous considerations about these AS, we recently conducted a study in industrial context[3] about the perception and management of GC, CD, UD and HL smells. Differently from the studies already introduced, in this case we did not validate Arcan detection results, but tried to gain a deeper and qualitative insight about the experience of practitioners in managing the presence of AS in their systems.

We collected data by interviewing 21 practitioners from 3 companies in Europe operating in two different domains (Embedded Systems and Enterprise Applications Development). The first company provided 12 participants, the second 6 and the third 3. The practitioners' background varies from a few years of activity (junior developers) up to 25 years of practice (architects). Interviews were semi-

3 A publication was extracted from this study [217], in collaboration with Darius Sas and Paris Avgeriou.

structured and each lasted approximately 30 minutes (see Appendix A.1.2 for the interview guide).

FEEDBACK ABOUT ARCHITECTURAL SMELL PERCEPTION Concerning the perception of the different kinds of AS, participants reported being the most familiar with GC among the four studied AS, because many of them reported personal experiences in managing this kind of smell. GC is perceived as a common cause of maintenance issues as well as reduced evolvability of the affected component, mainly as a result of the high level of complexity that characterizes its instances.

Opinions on CD were generally aligned, and most participants considered CD as detrimental for maintainability, reliability, and testability. Concerns about reliability (e.g. deadlocks) were mostly expressed by the participants working on C/C++ projects, highlighting that even if some CD instances have not caused issues yet, they pose a high risk for future undertakings. On the other hand, participants working with Java perceived it as less detrimental than other smell types like GC.

Opinions were much more polarized when the HL smell was discussed. Some participants mentioned that: (1) it should not be considered a problem because it could be a result of an intentional design decision; (2) it should not be a cause of concern as long as it is understandable; and, (3) as one participant expressed, it is easy to solve it. However, other participants (and especially the ones working with Java) mentioned that HL is very important to avoid because it is not easy to manage and it hinders both maintainability and evolvability by making it hard to understand how to insert new code in the presence of a HL. This feedback confirms the opinions of the developers described in the previous study (see Section 3.2).

Concerning UD, participants generally perceived it as a threat to both maintainability and evolvability, highlighting their concerns about the change ripple effects associated with UD and underlining the importance of avoiding dependencies towards packages that constantly evolve. Nevertheless, one developer expressed his doubts about the importance of this AS while few developers outlined that they did not fully understand it (they could not recall any similar experience and connect it to the UD definition) and gave no feedback about it.

FEEDBACK ABOUT ARCHITECTURAL SMELLS IMPACT ON MAINTENANCE AND EVOLUTION The participants discussed plenty of anecdotes and experiences about maintenance and evolution issues that they associated with the presence of AS. Almost all anecdotes about GC involve the difficulty of understanding the functionality provided by the component, mainly caused by the excessive internal entanglement of files (or classes), the excessive amount of function-

ality implemented, and the way functionality is scattered across the component. The relationship between GC and code duplications was also frequently discussed. Components affected by GC do not provide fine-grained classes that can be easily reused inside or outside the component, but large and entangled classes. Hence, when developers need to reuse an existing functionality, they prefer to copy the entire class and adapt it for the new purpose, instead of extracting a small, reusable functionality. On top of creating duplicated code, this also further enlarges the existing GC.

The experiences about CD are rather diverse and range from dealing with deadlocks and low throughput to unclear chain of command between components and poor separation of concerns in general. Cycles were also reported as an "intertwined mess" that is hard to understand; e.g. when there is a package that requests data from another package which in turn requests it back from the initial package. These problems resulted in a significant amount of effort required to be fixed or dealt with along the way, and in some cases they showed up only in production or at the customer. Participants also mentioned problems that had a more widespread impact; for example, a cycle prevented the creation of a microservice out of a subset of packages, as all the packages in the cycle had to be included in the microservice (the desired functionality could not be isolated, see Section 6.1.2).

Concerning HL, practitioners associated it with two types of issues: (1) difficulty of understanding the logic in the central component and (2) change ripple effects propagating from the components that the central component depends upon to the components depending on it, mentioning also a possible overlap with UD. The former was usually associated with how the central component exposes its functionality through its interface. The latter caused changes to unexpected parts of the system that practitioners did not expect to relate to the initial change, during activities such as bug fixing.

The maintenance issues that associated with UD the most, were change ripple effects. In several instances, practitioners reported that functional changes to a certain component (or package) also required several files in other components to change as well. As reported by two participants, the possibility of changes propagating to other components increases the difficulty of making changes: practitioners are forced to only make changes compatible with the other components in order to avoid changing and recompiling those other components.

FEEDBACK ABOUT THE INTRODUCTION AND MANAGEMENT OF ARCHITECTURAL SMELLS Participants reported their experiences about how they get to introduce an AS in the system. Some participants admitted that it often happens by design; for instance concerning GC, the component or the file is intended to be large. Subsequently, as reported by other interviewees, developers tend to un-

derestimate the severity of the introduced GC, while the incremental changes applied to it contribute in making it even larger. In other cases, AS are introduced inadvertently. For example, the participants reported that a bad separation of concerns at design time or the wrong exploitation of class inheritance, can result in CD. Another participant mentioned that they used to create a dedicated interface to hide unstable components behind it as a "practice" to avoid the propagation of changes; however, this is precisely the description of a UD smell, being misinterpreted as a good practice. In many cases, introducing AS seems unavoidable and accepted as a "necessary evil". For example, one participant explained that in view of an imminent deadline, they focus on developing the new feature and having a first structure of the code, without caring about its maintainability. Moving on to the management of AS, we asked the participants about their experiences with AS refactoring. Most of them had experience with the refactoring of GC, in particular the practice of splitting the component in smaller pieces by applying incremental changes or by detaching the smallest, easiest sub-components first. One interviewee managed to break a CD by re-modelling the involved dependencies to follow a hierarchical structure; others reported creating replacement interfaces and slowly migrating clients to them while refactoring the existing components. In contrast, developers do not commonly refactor HL because of the required effort; if they can, they tend to "code around it" without removing it when developing new features, allowing it to persist. One interesting reason mentioned for not refactoring AS is the absence of a comprehensive regression test suite. Concerning practices which support the refactoring of AS, some participants mentioned the usage of SonarQube to keep the code readable and maintainable; this can ease the refactoring of AS, since often the poor quality of the code makes refactoring even more difficult and time-consuming. Another indicated pair programming and the help of senior developers as valid support. However, not all the interviewees reported the adoption of refactoring practices. Some even pointed out that they avoid refactoring because their clients do not pay for refactoring time and as long as the system has no visible problems in production, they do not intervene.

Finally, we also asked whether practitioners use tools to manage architectural smells. SonarQube was mentioned by quite a few participants, but only once in regard to an AS (i.e. to detect cycles). Besides that, practitioners do not rely on any specific tool to manage AS.

FEEDBACK ABOUT SCATTERED FUNCTIONALITY AND FEATURE CONCENTRATION We extended the study with 8 of the 21 developers, in particular 4 junior developers, 2 senior developers, one DevOps specialist and one Scrum Product Owner, working on enterprise Java projects. We distributed an online survey to investigate their opinion,

specifically about the perceived harmfulness of Scattered Functionality and Feature Concentration. From their answers, all the smells have an impact on software quality. In particular, both SF and FC impact readability and maintainability. SF is a problem because *"Makes the code less reusable"* and a *"problem arises when the change to a feature involves multiple components."*. Moreover, *"SF is difficult to maintain over time and new changes [when present] degrade the system quality and its efficiency"*. FC should be avoided because *"Every class should have only one responsibility and should not cover multiple concerns"*.

3.4 SUMMARY OF THE FINDINGS

We now resume the results of our research about Arcan validation and the perception of AS by industrial developers. Table 3.2 resumes the interesting feedback we had from the practitioners who we reached out. For each AS detected by Arcan, we indicate an example of false positive AS (*False positive example* column) encountered by the developers; the general perception of the smells along with the impact they have on the affected system (*Perception and Impact*); finally the quality attributes which are affected by the smells (*Affected Q.A.*, legend at the bottom of the table).

For what concerns Arcan validation and AS false positives, we acknowledge that we are far from the 100% precision of the results, in terms of identifying smells that are considered as *real problems* by developers. The false positives we identified during our validations are not easy to spot, because they require the detector to capture factors external to code. For instance, in some cases developers declared they introduced the smells *by design*, meaning that they were aware of what they were doing. In other cases, developers admitted that *they had no other choices*. Both static and dynamic analysis and even the exploitation of Natural Language Processing (NLP) techniques cannot extract this kind of information from code. To overcome this problem, the detection approach must be fundamentally different and involve the developers' input in the first place. Some works in this direction have been conducted in the field of self-admitted technical debt [201], where the symptoms of debt are searched in the code comments and in other natural language bits, such as messages of git commits. Other research lines point to annotate the code with heterogeneous information, such as annotation about architectural decisions [161] and design flaws [191]. In brief, future works about AS detection and the management of false positive smell instances should focus on developing approaches able to collect architectural decisions information.

Concerning the perception of AS, i.e., what developers think about the smells independently from the tools able to detect them, we can confirm that developers actually experience AS and their consequences in everyday coding and that the six considered AS, when actually

present, represent a problem for the maintainability of the code. However, even if we found agreement, the perception, as implied by the term itself, is subjective and bounded to the personal opinions of the single developer, determined by his/her/they past experiences, seniority, education and skills. The smells for which we noted the majority of discordance are Cyclic Dependency and Hub-Like Dependency. To conclude, architectural smells are real problems, causing concrete shortcomings and worthy of attention by the software engineering community. Architectural smells are inherently complex, because of the fact they are *design decisions*: decisions cannot be inferred directly from code and depend from many external factors, and thus they cannot be generalised. Instead, they are tied to the people who are making such decisions: software architects and developers. Future directions concerning this subject can be found at the end of the thesis in Section 8.2.

Table 3.2: Summary of architectural smells perception

AS	**False positive example**	**Perception and Impact**	**Affected Q.A.**
CD	Callbacks from anonymous classes in GUI components, considered a necessary solution.	Creates a lot of side effects when refactored.	Maintainability, Reliability, Testability
FC	-	Hinders the migration to microservices.	Maintainability and Readability
GC	An intentional design choice.	Very dangerous. Internal entanglement prevents changes. Hinders the migration to microservices. The affected code is hard to understand.	Maintainability and Evolvability
HL	A package containing utility classes, created by design.	It get worse as long as time passes. It is the worst smell to have in a system.	Maintainability, Evolvability and Understandability
SF	An architecture designed by layer (one layer per package).	Makes thecode less reusable. Hinders the migration to microservices.	Maintainability and Readability
UD	Sometimes caused by the presence of the Strategy design pattern.	It should be avoided. Not fully understood by developers.	Maintainability and Evolvability

4

EMPIRICAL STUDIES ON ARCHITECTURAL SMELLS

The development and validation of Arcan gave us the opportunity to analyse many software projects and conduct empirical studies on architectural smells. In particular, the studies presented in this chapter regard the evolution of AS and their correlation with AS themselves and design patterns.

The general workflow of such studies consists in *1)* the definition of the research questions *2)* the collection of raw data, usually the output of one or more software analysis tools executed on public software repositories; *3)* the generation of a dataset; *4)* the execution of statistical tests to investigate possible relationships among the data; *5)* the interpretation and discussion of the results, with a focus on the insights and takeaways that could be useful for the software engineering research community and the software developers.

A common threat for such kind of empirical analysis is the difficulty of doing validation of the results: the ideal validation would be to submit the studies' outcome to the developers or to compare the outcome with a ground-truth. However, developers are not easy to reach out and unfortunately, to the best of our knowledge, there are no available ground-truths containing data about AS. In any case, the studies reported in the following sections comprise also a discussion of their threats, and when possible we complemented the data analysis with manual validation, by checking the results (e.g. the correlation between a smell and a design pattern) directly in the source code of the analysed projects.

In the following sections, first we introduce the main statistical tests and techniques we exploited to analyse architectural smells data, then we describe one by one our empirical studies. At the end of the chapter, we summarise our results and findings.

4.1 EXPLOITED STATISTICAL TESTS AND TECHNIQUES

Given a large quantity of data, we need tools and techniques to extract valuable information, on the top of which we can draw some conclusions about the research questions defined in our empirical studies. Thus, we run well-known statistical tests, implemented in popular languages (R language [241]) and tools (KNIME platform [114]). We know introduce them and explain how we use them in the context of our empirical studies.

4.1.1 *Correlation analysis*

Correlation analysis is a method of statistical evaluation used to study a relationship between two variables. Measuring the correlation between two dimensions can be useful to understand whether they are in some kind of relationship. When testing the correlation of variables such as number of architectural smells and number of design patterns, we relied on Pearson [34], Spearman [227] and Kendall [119] correlation tests. Spearman *rho* and Kendall *tau* rank correlations are two of the correlation coefficients commonly used to measure the strength of the relationship between two variables that are not normally distributed. In our study we usually adopt them because our variables tend not to be normally distributed. This because synthetic (not physical) quantities such as the properties measured on software (number of lines of code, coupling, cyclomatic complexity, number of architectural smells) do not follow the normal distribution, but the power law [146]. In any case, we checked the normality of the considered variables in each of our studies. The correlation coefficients resulting from the tests ranges from -1 to 1, where -1 indicates strong negative correlation, 1 is strong positive correlation and 0 means no correlation. The result is considered significant if the p-value of the test is < 0.05.

4.1.2 *Principal Component analysis*

Principal Component Analysis (PCA) is a multivariate technique for identifying the linear components of a set of variables. In our context, we exploit PCA to identify relationships among several variables (e.g. architectural smells), not just pairs, as done by correlation analysis. For instance, it can be useful to detect collocation between set of three or more AS. Some previous studies used this analysis with the same aim, but to identify collocated code smells Similar usage of PCA has already been done in previous studies to identify collocated code smells [253, 254]. To check whether the data are suitable for PCA, as suggested in the work of Walter et al. [254], we compute the Kaiser−Meyer−Olkin measure (KMO) [115]. This test measures sampling adequacy for each variable in the dataset and for the complete dataset. It is a measure of the proportion of variance among variables that might have common variance. The lower the proportion, the more suited the data is to PCA. Quality of the data sample is claimed satisfactory if KMO > 0.50 [115]. Still following Walter indication, we also run the Bartlett's test of sphericity [31], which verifies if the dimensionality of the dataset can be effectively reduced. The two described tests are complementary and in our studies we run both of them to assess the suitability of the dataset for PCA analysis.

4.1.3 *Association rules extraction*

An association rule is the expression of a relationship among data items in a dataset. It is an *if-then* statement composed by an `{antecedent}` and a `{consequent}`. An example of a rule explaining the relationship between two AS is `{HL, FC}` $\rightarrow$ `{CD}`, which can be read as "*If* Hub-Like Dependency and Feature Concentration affect a component at the same time, *then* the component belongs to a cycle". The association rule extraction technique aims to automatically extract such rules from a dataset composed of *transactions*; in our case a transaction corresponds to a vector of the binary features (architectural smells, software metrics etc.) associated to a single component.

We also adopt commonly used metrics for evaluating a quality of a rule: support, confidence [4], conviction and lift [42], with the following definitions.

Given a rule defined as $X \rightarrow Y$,

Support (*Supp*) of a rule is the ratio of transactions that match the rule with respect to the entire dataset.

Confidence (*Conf*) is the ratio of transactions that contain both the antecedent X and the consequent Y.

$$Conf(X \rightarrow Y) = \frac{Supp(X \cup Y)}{Supp(X)} \tag{7}$$

Lift is the ratio of the observed support to the expected support, if X and Y are independent. If lift is equal to 1, it means that the rule is not significant for the dataset.

$$Lift(X \rightarrow Y) = \frac{Supp(X \cup Y)}{Supp(X) \times Supp(Y)} \tag{8}$$

Conviction (*Conv*) is the ratio of the probability that X will appear without Y if they are dependent, divided by the observed frequency of the appearance of X without Y. Conviction is useful to measure the degree of implication of the association, i.e., how much Y depends on X; in particular, high conviction indicates that the consequent is highly dependent on the antecedent, while conviction of value 1 means that the items are unrelated.

$$Conv(X \rightarrow Y) = \frac{1 - Supp(Y)}{1 - Conf(X \rightarrow Y)} \tag{9}$$

4.1.4 *Mann-Kendall test*

We exploit the *Mann-Kendall test*, which is a non-parametric test able to assess if there is a monotonic upward or downward trend of a variable of interest over time. The null hypothesis for this test is that there is no monotonic trend in the series. The alternate hypothesis is that a trend exists. This trend can be positive, negative, or non-null.

Notice that this test can be used to find trends for as few as four samples. In our case, usually one sample corresponds to one code commit (version). However, with only a few analysed samples, the test has a high probability of not finding a trend when one would be present if more commits were provided. Hence, in our studies we analyse at least eight commits.

4.2 A STUDY ON CORRELATIONS BETWEEN ARCHITECTURAL SMELLS AND DESIGN PATTERNS

Design patterns (DP) [88] are generic, reusable solutions for recurring software design problems. Their adoption is widely recommended, since they capture verified, distilled knowledge based on experience. They also explicitly identify the trade-offs between their advantages and shortcomings, which help developers in making informed, conscious decisions concerning software design.

The two concepts, Architectural Smells (AS) and design patterns seem not only unrelated, but even also disjoint, as they represent two fundamentally different approaches to software quality. DP provide *recommendations* for software design issues, securing some additional quality properties, like flexibility, reusability or extensibility. They are directly applicable, as they include instructions for implementation. On the other hand, AS represent *warnings* that indicate the possible presence of deeper quality issues that cannot be directly identified or whose identification is hindered. Additionally, DP and AS refer to different levels of abstraction: while DP address the *tactical* level, solving problems with a limited scope of methods and classes, AS usually comprise more comprehensive issues involving modules, packages or components, having a more *strategic* impact on the entire system. As a result, even if AS and DP may be collocated or related at the structural level, there is currently no evidence that their interactions have an impact on the quality.

MOTIVATION However, a more thorough examination reveals some scenarios in which the *relationships* between DP and AS may affect the system in meaningful ways, reaching far beyond their individual impact. In this context, by "relationship" we mean a co-occurrence of DP and AS within the same software dependency connecting two architectural components (e.g., Java classes and packages).

First, the use of a DP is a deliberate decision of a programmer to apply a specific solution to a given problem. Each pattern has predictable consequences, showing the benefits and trade-offs of its application. However, they can be reduced, changed or even reversed due to interfering factors that change the structure or behaviour of the pattern [11]. For example, the advantages of applying the Template Method design pattern to structure inheritance hierarchies are

diminished if the respective methods overriden in the subclasses do not follow the Liskov Substitution Principle [145] or are affected by a Tradition Breaker Code Smell (CS) [128]. Similarly, an unidirectional dependency structure in the Observer pattern could be broken by introducing cycles to it [77]. CSs have already been found to interact with patterns [253][10], thereby affecting their prevalence. Moreover, other studies found a relationship between the evolution of software architecture and design pattern decay[109][76][75], i.e., architectural changes may break the structural or functional integrity of a design pattern. Therefore, we may conjecture that various AS, frequently referring back to an architectural planning phase, can also impact, restrict or even prevent the application of specific DP.

Secondly, the relationship between AS and DP can also be reverted: smells can be manifestations of defects in pattern instances [165], e.g., a flawed Chain of Responsibility DP with two-directional dependencies will result in the Cyclic Dependency AS. In that case, effort invested in removing the smell can also fix the pattern implementation. In general, while several studies found DP to have a positive impact on software quality, other works reached the opposite conclusion. The use of patterns does not always result in fewer defects, for instance, Singleton and Observer appears more defect-prone than others [250]. Moreover, the adoption of design patterns increase the maintenance effort, because managing code containing patterns requires more time than implementing pattern-unrelated solutions [176][202]. Some DP (Composite, Abstract Factory and Flyweight) have a negative impact on the reusability and understandability of code[120]. Finally, classes not involved in DP, or involved in some complex and change-prone patterns, e.g., Decorator and Template Method, can be more prone to violations [75][258].

We propose a final observation. Smells are manifestations of deeper design or architectural issues, but this relationship is inevitably affected by uncertainty: not every smell refers back to a real underlying problem. As a result, in some cases they can be conscious and accepted effects of a DP, which would make them false positives. This effect has been already studied with regard to code smells: the presence of a smell can be attributed to the application of a DP [77]. At an architectural level, Cyclic Dependencies smell [19], describing circular references among components, can be an effect of implementing callbacks [231], a common notification mechanism in GUI-related applications. Also a Hub-Like Dependencies AS, describing architectural components with numerous dependencies, can be incorrectly detected, when actually being the implementation of Controller or Orchestrator patterns [84].

Following from these observations, we can conclude that the effects of the mutual interactions between an AS and a DP may be non-trivial and multi-aspect, and we can expect their significant im-

pact on selected quality characteristics. For that reason, the study of these relationships deserves a closer analysis to determine how and to what extent these concepts affect each other. Although a link between code smells and DP has been already analyzed [111, 226, 253], no empirical study on AS has been presented. The aim of our study is to investigate this subject[1].

Since several AS concern dependencies between architectural components, and DP strictly influence the design of such dependencies, we can hypothesize that the eventual relationship between the two phenomena will be revealed by analysing these dependencies.

For this reason we provide and use a dataset of 60 open source Java projects which reports classes and packages having AS and DP, together with the dependency information. The dataset has been created with two existing academic software analysis tools: *Pattern4* [244] for DP detection and *Arcan* (see Chapter 2) for AS detection.

The results of our study can draw developers' attention to the key parts of software systems, and can enable them to incorporate knowledge of the extracted AS-DP relationships.

The main contributions of this study are three-fold:

- we provide *a dependency dataset*, with information regarding object-oriented dependencies, AS and DP;
- we present *a study on the frequency and correlation of AS and DP in 60 open source projects*, based on statistical analysis, e.g., correlation analysis and mining association rules;
- we formulate *useful hints for developers and researchers*, to help them avoiding potentially hazardous combinations of DP and AS, as well as to enhance the detection strategies for AS.

4.2.1 *Empirical Study Design*

In the following section, we describe our research questions which guide the study of AS and DP relationship. In particular, we aim to *1)* investigate the frequency of both phenomena in the analyzed projects, and *2)* understand whether there are specific pairs of AS and DP frequently involved in a relationship.
Our study aims to answer the following research questions:

- *RQ1*: *What is the distribution and prevalence of AS and DP in Java projects?* We want to understand how many components (class or package) are affected by AS and DP. This is useful for researchers and developers to have an overview about the frequency of the two phenomena, and also for us to set the stage

1 A publication was extracted from this study [199], in collaboration with Bartosz Walter.

for the other research questions. In particular, we consider two specific issues:

RQ1.1: *Is there a difference in the distribution of AS and DP with respect to the considered projects?* Rationale: We aim to analyze how both AS and DP are distributed in the considered software projects. Our aim is to identify the most frequent types of smells in software projects, so that the developers can focus their attention on them.

RQ1.2: *Is there a difference in the distribution of AS and DP with respect to various application domains?* Rationale: This question helps in assessing to what extent specific application domains are affected by AS and DP. It is important to know which domain is more open to AS, so that developers can be more aware of AS when dealing with projects belonging to it. It is important also to compare the presence of AS with the presence of DP, to understand if domains interested by many DP are also the ones with fewer AS.

- *RQ2*: *Which DP-AS pairs display significant relationships?* Rationale: A strong relationship among a specific type of AS and DP may indicate that the implementation of a pattern is a root cause for the introduction of a smell, meaning that those DP, contrary to their purpose, have a negative impact on quality. However, we could discover that some AS imply the presence of some DP. This could mean that those AS are false positives i.e., the smell is present because developers had intended to program it. On the other hand, it could signify that certain DP can help in mitigating the negative impact caused by the AS, and they are employed by developers to remedy the problem.

- *RQ3*: *Can the presence of AS imply the absence of DP?* or vice-versa *Can the presence of DP imply the absence of AS?* Rationale: The presence of the considered AS in the system affects the dependency structure of the system itself. DP are implemented by manipulating the structure of the system and the implementation is possible when the structure respects some principles (e.g. it must be acyclical). Hence we want to investigate if the presence of specific AS results in the absence of specific DP, and vice-versa. This would reinforce the conclusion that AS and DP are mutually exclusive concepts.

ANALYZED PROJECTS AND COLLECTED DATA In this study we analyzed the 60 open source Java projects presented in Table 4.1. This is a subset of projects being curated under Qualitas Corpus (QC) [238], and their selection was conditioned by the availability of properly compiled code, which is necessary for the AS detector. For the projects, we report their application domain (Domain), name

(Project), analyzed version (Version), number of classes (NOC), number of packages (NOP), and the total number of lines of code (LOC). These projects have diverse characteristics: they are assigned to seven different domains (Graphics, Database, IDE, Middleware, Parser, Testing, Tool), have different sizes (ranging from 2809 to 651118 LOC), and are developed by different open source communities (Apache, Eclipse, etc.). In order to balance the dataset, the original domains defined in QC have been adjusted: 3D/Graphics/Media, Diagram Generator/Data Visualization and Games have been named "Graphic", and "SDK projects" were merged with "IDE" as "IDE".

Table 4.1: Analyzed Projects

Domain	Project	Version	NOC	NOP	LOC
Database	axion	1.0-M2	257	13	24163
	cayenne	3.0.1	2991	185	192431
	db-derby	10.9.1.0	3010	217	651118
	hsqldb	2.0.0	644	26	143870
	squirrel-sql	3.1.2	73	2	6944
	hibernate	4.2.0	7119	856	431693
Graphic	batik	1.7	2299	81	178469
	displaytag	1.2	320	32	20498
	drawswf	1.2.9	311	34	27674
	itext	5.0.3	583	34	78348
	jasperreports	3.7.4	1709	61	169821
	jext	5.0	761	59	60160
	marauroa	3.8.1	247	41	17733
	megamek	0.35.18	1859	37	242836
IDE	checkstyle	5.6	533	42	36641
	colt	1.2.0	381	24	35919
	drjava	stable-20100913-r5387	1210	30	89477
	eclipse SDK	3.7.1	24871	1425	2484311
	jpf	1.5.1	140	10	13342
	nakedobjects	4.0.0	2975	496	133936
	trove	2.1.0	72	4	5845
Middleware	informa	0.7.0	223	26	13874
	jena	2.6.3	1279	48	65774
	jspwiki	2.8.4	582	70	60250
	jtopen	9.4	1915	15	342032
	openjms	0.7.7-beta-1	616	66	39435
	oscache	2.3	115	22	7624
	picocontainer	2.10.2	206	15	9253
	xmojo	5.0.0	22	9	2809
	quartz	1.8.3	269	51	28557
	QuickServer	2.1.0	196	28	18339
	sunflow	0.07.2	209	22	21970
	tapestry	5.1.0.5	2119	139	97206
Parser	ant	1.8.2	1608	122	127507
	antlr	3.4	381	20	47443

Continued on next page

Table 4.1 – *Continued from previous page*

Domain	Project	Version	NOC	NOP	LOC
	apache-maven	3.0.5	837	143	65685
	javacc	5.0	107	8	14633
	jparse	0.96	75	4	24796
	nekohtml	1.9.14	64	7	7647
	xalan	2.7.1	1402	86	183709
	xerces	2.10.0	947	53	125973
Testing	cobertura	1.9.4.1	160	34	54555
	emma	2.0.5312	290	27	21492
	findbugs	1.3.9	1432	67	110782
	fitjava	1.1	95	5	3457
	jmeter	2.5.1	1038	175	94778
	junit	4.10	171	28	6580
	log4j	2.0-beta	606	61	32658
	pmd	4.2.5	872	88	60739
Tool	freecs	1.3.20111225	146	12	22645
	heritrix	1.14.4	656	48	64916
	james	2.2.0	306	31	27087
	jfreechart	1.0.13	1037	69	143062
	jgraph	5.13.0.0	298	34	31818
	jgraphpad	5.10.0.2	375	22	24208
	jmoney	0.4.4	83	4	8197
	jsXe	04_beta	251	14	18494
	pooka	3.0-080505	491	28	44474
	proguard	4.9	648	35	62618
	webmail	0.7.10	115	19	10147

We collected the data on three architectural smells, namely Cyclic Dependency, Hub-Like Dependency and Unstable Dependency (see Section 2.2), and all the DP described in Table 4.2. These patterns have been defined by GoF [88], except for *Proxy2*, which is a variation of the Proxy pattern. In that case, also called *Dynamically-Typed Proxy* [35], the Proxy role has an association to Subject role (named subject) and the method `Request()` declared in Proxy invokes an abstract method having the same signature through the Subject association. We chose the reported patterns since they are all identified by one tool, called *Pattern4*, and represent a different subset of patterns proposed in the GOF catalogue [88].

We performed our analysis on different aggregations of data: project data, application domain data, and the entire dataset. By the "*granularity level*" we mean the specific type of a Java component: class or package.

TOOLS To detect DP we used **Pattern4** [244], capable of extracting the patterns from the analysis of Java project's static structure. In particular, the implemented detection methodology is based on similarity scoring between graph vertices, where a graph represents the

project under analysis. We decided to use this tool due to its free availability and the large number of detected DP. Moreover, it has been validated on 3 open source projects, having very high (95%-100%) precision and recall [244].

To detect AS, we employed **Arcan** (see Chapter 2).

Table 4.2: Detected Design Patterns

Name	Type	Description
Factory Method (FM)	Creational	Define an interface for creating an object, but let subclasses decide which class to instantiate.
Prototype (P)	Creational	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton (S)	Creational	Ensure a class only has one instance, and provide a global point of access to it.
Object Adapter (A)	Structural	Convert the interface of a class into another interface clients expect.
Composite (C)	Structural	Compose objects into tree structures to represent part-whole hierarchies.
Decorator (D)	Structural	Attach additional responsibilities to an object dynamically.
Bridge (B)	Structural	Decouple an abstraction from its implementation so that the two can vary independently.
Proxy (PR)	Structural	Provide a surrogate or placeholder for another object to control access to it.
Proxy2 (PR2)	Structural	Proxy variation reported by Gunter Kniesel and Alex Binun from University of Bonn
Command (COM)	Behavioural	Encapsulate a request as an object thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Observer (O)	Behavioural	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State (ST)	Behavioural	Allow an object to alter its behavior when its internal state changes.
Strategy (STR)	Behavioural	Define a family of algorithms, encapsulate each one, and make them interchangeable.
Template Method (TM)	Behavioural	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
Visitor (V)	Behavioural	Represent an operation to be performed on the elements of an object structure.
Chain of Responsibility (COR)	Behavioural	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

DATASET We now introduce the dataset used during the analysis, named *Dependency Dataset*. It has been created in response to the observation that in object-oriented projects both AS and DP affect the

structure of the code, specifically the dependencies between classes/-packages. Moreover, for DP, the direction of the dependencies and their type (class dependency, inheritance dependency and interface dependencies) are important for defining the pattern itself. Arcan is able to represent both classes/packages (nodes) and dependencies (edges) in the dependency graph. Hence, the dataset built and exploited for this work is *edge-based*[2].

With the **Dependency dataset** we are able to model each dependency in the dependency graph and determine if the dependency is part of an AS and/or a DP. The dependencies that Arcan can extract belong to one of two granularity levels: class or package. There are four types of dependencies considered: *classDependency* between class A and class B, if there is at least a method call from A to B; *inheritanceDependency* between class A and class B, if A extends B; *implementationDependency*, between class A and interface B, if A is an implementation of B; *packageDependency*, between package C and package D, if a class in C has a (class/inheritance/implementation) dependency with a class in D.

The *Class dependencies* dataset, whose features are reported in Table 4.3, provides information about:

- the *head* of the edge, i.e., the name of the class from which the dependency originates;
- the *tail* of the edge, i.e, the name of the class at which the dependency ends;
- the *type* of dependency. Three types are considered: *classDependency*, *inheritanceDependency* and *implementationDependency*.
- the *weight* of the edge (the number of times the dependency is realized in the code);
- the smells (*AS1 ... AS#*) and the design patterns (*DP1 ... DP#*) that involve the dependency. AS and DP are considered binary features (we count the presence/absence of AS and DP in the dependency).

Table 4.3: (Class) dependency dataset features

head	tail	type	weight	AS1	AS2	...	AS#	DP1	DP2	...	DP#

The *package dependencies* dataset stores the same features except for the "type"; as for packages we consider only *packageDependency*. Moreover, the detected AS for classes are (Class) Cyclic Dependency and (Class) Hub-Like Dependency; the detected smells for packages

2 Replication package is available at https://drive.google.com/drive/folders/1ONSTAwyvK9d7gGp70kgLXfDZvkU80xz1

are (Package) Cyclic Dependency, (Package) Hub-Like Dependency and Unstable Dependency. Since DP are structures implemented at the class level, we had to aggregate our data at the package level. We say that a package dependency is involved in a DP, if the corresponding dependency at the class level (i.e., a cross-package class dependency) is involved (see Figure 4.1). For instance, if package A depends on package B (A $\rightarrow$ B) and class C1 that belongs to A depends on class C2 of package B (C1 $\rightarrow$ C2), then all the DP involving (C1 $\rightarrow$ C2) are also counted for (A $\rightarrow$ B).

This dataset allows for counting:

- The number of dependencies involved in at least one AS,
- The number of dependencies involved in at least one DP,
- The number of dependencies involved in an AS and a DP,
- The number of dependencies not involved in any AS,
- The number of dependencies not involved in any DP,
- The number of dependencies not involved in any AS or DP.

ANALYSIS Our dataset consists of two levels: the **1) class level**, with *data point*: Java class dependency, and *features*: DP and AS data; **2) package level**, with *data point*: Java package dependency, and *features*: DP data of the package dependencies, and AS data. We consider (Class) Hub-Like Dependency and (Class) Cyclic Dependency smells, (Package) Hub-Like Dependency, (Package) Cyclic Dependency and (Package) Unstable Dependency smells.

Given this representation, we used three analysis techniques in order to answer our RQs related to the frequency of AS and DP and the relationships among them.

- **Comparison of DP and AS frequencies** In order to answer *RQ1.1 and RQ1.2*, we computed the absolute and relative frequencies of both AS and DP at the class and package level.

  Used technique: distribution statistics.

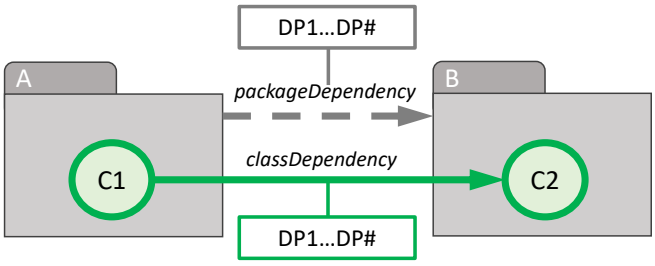


Figure 4.1: Aggregation of DP from class level to package level

- **Correlation analysis** In order to answer *RQ2 and RQ3* we performed the Spearman [227] and Kendall [119] correlation analysis on the entire dataset. We chose them because we checked the normality of our variables and discovered they were not normal. In particular, we ran the Anderson-Darling test [172] for normality. We could not use the well known Shapiro-Wilk test [219] because our variables exceeded the maximum number of data-points (> 5000) allowed in input by the test. Moreover, we could not use the Kolmogorov-Smirnov test, since it is not suitable when estimating the parameters from the data (as we do), and it also expects a continuous distribution that does not contain any ties (repeated values). Instead, the Anderson-Darling test does not require the mean and the standard deviation to be supplied. Additionally, we exploited Q-Q plots [260]. A Q-Q plot is a graphical method for comparing two probability distributions by plotting their quantiles against each other. These plots are often used when the dataset is large enough to introduce bias in the Shapiro-Wilk test, as in our case.

  Used technique: computation of *Spearman's rho* and *Kendall's tau* rank correlation coefficents. *Used tool*: R language `cor()` function [241].

- **Association rules extraction** Moreover, related to answer *RQ2 and RQ3* we aim to exploit association rules to identify relationships among AS and DP. An example of a rule for the AS and DP dataset is `{HL, Singleton}` $\rightarrow$ `{CD}`, which can be read as "*If* Hub-Like Dependency and Singleton pattern affect a dependency at the same time, *then* the dependency belongs to a cycle".

  Used technique: an implementation of the Apriori algorithm [5]. *Used tool*: `apriori` function from the `arule`[3] R package. *Parameters*: for all datasets, we fixed the *minimum support* to 0.001 and we reported the rules with *confidence* $\geqslant$ 0.6, as used in other empirical studies [254]. In the following, we provide the details concerning the three analyses conducted on both classes and packages.

4.2.2 *Results*

This section reports the results of our analysis starting from (1) the computation of statistical information, to answer RQ1.1 and RQ1.2, followed by (2) correlation analysis, and (3) association analysis to answer RQ2 and RQ3. The complete results can be found in the replication package.

3 https://cran.r-project.org/web/packages/arules/index.html, accessed October 2021

Table 4.4: Descriptive statistics for the dependency dataset

	Tot.	Min	Max	Mean	Median	St. dev.
class dependencies						
#dependencies	226066	44	32834	3767.767	1692	5467.270
#not AS/DP	139416 (62%)	27	21054	2323.600	863.5	3574.855
#AS	43420 (19%)	8	4687	723.667	266	998.051
#DP	31883 (14%)	0	7578	531.383	240	1083.069
#AS/DP	86650 (38%)	17	11780	1444.167	571	2058.908
package dependencies						
#dependencies	17362	1	4450	289.367	102	685.826
#not AS/DP	5341 (30%)	0	1860	89.017	23	276.486
#AS	6036 (34%)	0	1101	100.600	65	455.039
#DP	2084 (12%)	0	804	34.733	8	114.867
#AS/DP	12021 (69%)	0	2590	200.350	79	413.653

AS AND DP DISTRIBUTION AND PREVALENCE RESULTS Starting with the answer to RQ1, we provide a description of the results regarding how much the different types of AS and DP affect the analyzed projects at the two studied granularity levels.

Table 4.4 reports the frequency of AS and DP at class and package level, without considering their types. Tables 4.5 and 4.6 report the results for AS and DP, respectively, at the class and package level, depending on the different types of AS and DP. The tables indicate data extracted from all the 60 projects, and for each reported quantity we provide the *total*, the *minimum*, the *maximum*, *mean* and *median* values. In particular, Table 4.4 shows some aggregated statistics for the entire dataset. Each row reports the number of analyzed dependencies (#dep.), the number of dependencies not involved in any AS or DP (#not AS-DP), the number of dependencies *only* affected by AS (#AS), the number of dependencies *only* involved in DP (#DP) and, finally, the number of dependencies involved in AS *or* DP (#AS-DP). The percentage values reported in the parenthesis of Table 4.4 are in relation to the total number of dependencies. The percentages reported in Tables 4.5 and 4.6 do not sum up to 100%, because some dependencies are affected by more than one smell or design pattern at the same time.

In the following, we introduce a detailed analysis of our dataset by (1) investigating the frequency of AS and DP in the 60 Java projects and their domains, (2) reporting the AS statistics and (3) the DP statistics.

Comparison of AS and DP frequencies Figures 4.2 and 4.3 show the frequency of AS and DP in the subject projects at the class and package granularity levels. The x-axis indicates the projects ordered by ascending number of dependencies, and the y-axis shows the number of AS and DP for each project (highlighted in two different colors). As we can see from the diagram, the trend is growing for both AS and DP.

By inspecting the proportion of AS and DP with respect to the different project domains (Figure 4.4 and 4.5), it becomes clear that the projects that present high disparity in the proportion of AS and DP belong to the *Database* domain. In particular, the number of dependencies affected by AS at the class level is 10900, while by DP it is 15712; 3927 and 8570 at package level. Through Figure 4.6 and 4.7 we can understand which specific DP is most prevalent, along with the highest concentration of AS and in which domain.

> In general Template Method, Singleton and Factory Method are the most frequent DP, at both class and package level and in each domain.

However, AS are more concentrated in selected domains, such as Parser and Testing at class level, Graphic and Parser at package level. By merging together the two indicators, DP presence and AS concentration, the following patterns are the most frequent, with the highest concentration of smells: Template Method, Singleton and Factory Method in Parser and Testing domains at class level; Template Method, Singleton and Factory Method in Database and IDE domains at package level.

Architectural smells statistics Concerning RQ1.1, and in particular the frequency AS, Table 4.5 reports the frequency of AS in the 60 analyzed projects, considering their type (*CD*, *HL*, *UD*) and granularity level (class and package). In particular, we indicate the total, minimum, maximum and mean number of the instances of each specific type. The most frequent smell at both class and package level is CD. It is also the only smell which is present in all projects: in fact, the minimum value of detected CD is 4, meaning that there is at least one project with at least 4 cycles. At the package level, HL is less diffused than CD.

Concerning RQ1.2, we now report the diffuseness of AS in relation to the application domains of the analyzed projects.

Figure 4.8 and 4.9 show the frequency of class and package AS over the 7 different domains.

> The most diffused smell, for both the granularity levels, is Cyclic Dependency (CD).

At the class level, its presence is almost equal in all the domains, with $mean = 7422$.

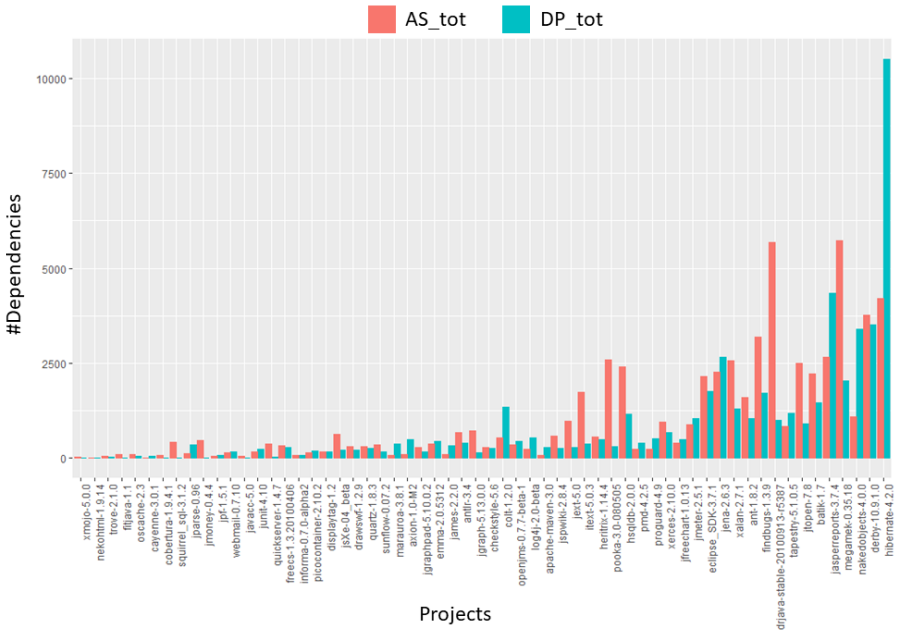


Figure 4.2: Frequency of class level AS and DP in 60 Java projects

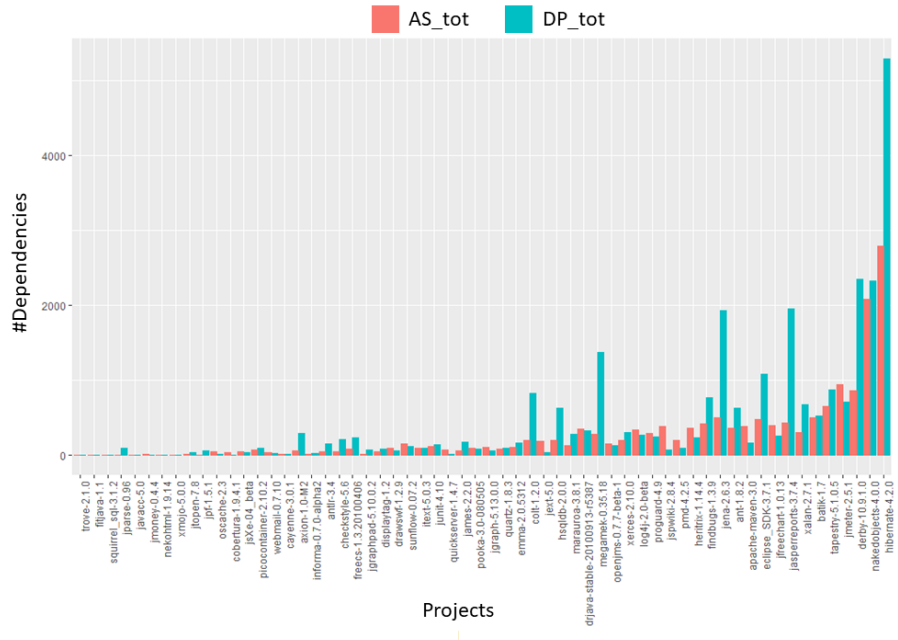


Figure 4.3: Frequency of package level AS and DP in 60 Java projects

The most affected domain is Database, with 10149 CD instances. However, the number of CD instances varies depending on the different domains at package level. Graphic, Middleware and Parser domains have the lowest number of CD instances, while the Testing and Tool domains have a medium number of the smell and, finally,

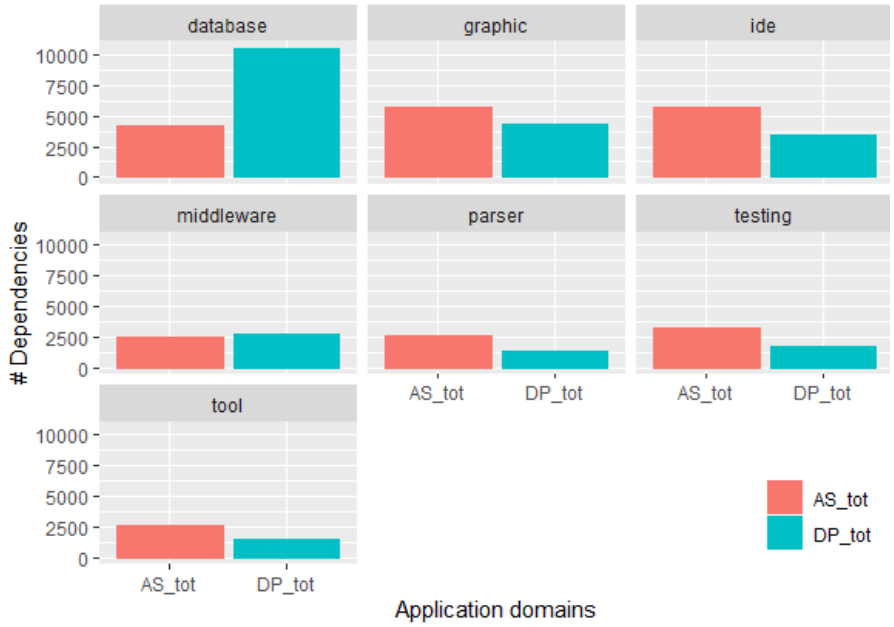


Figure 4.4: Frequency of AS and DP in 7 domains - Class level

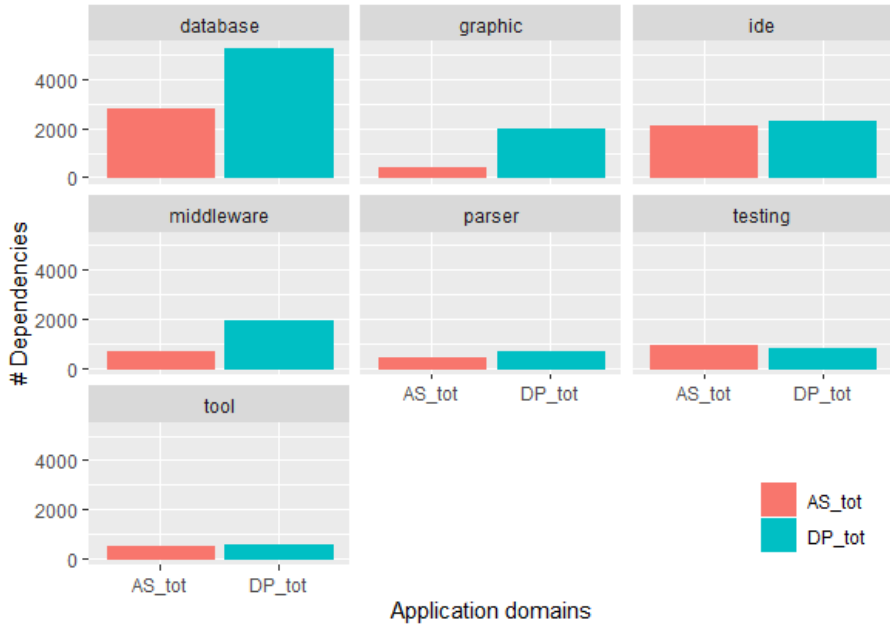


Figure 4.5: Frequency of AS and DP in 7 domains - Package level

Database and IDE are strongly affected by CD, with more than 1500 smells.

Concerning Hub-Like Dependency (HL), the most affected domain at the class level is IDE, while at the package level it is Middleware. For the Unstable Dependency (UD) smell, Database is the most affected domain, with 1246 instances.

Table 4.5: Statistics for architectural smells in the dependency dataset

AS	Tot.	Min	Max	Mean	Median	St. dev.
			class dependencies			
CD	51959 (94%)	4	5504	865.983	281.5	1213.474
HL	8301 (15%)	0	1477	138.35	63.5	240.8923
			package dependencies			
CD	8445 (84%)	0	1627	140.750	63	266.9363
HL	2780 (27%)	0	259	46.333	29	55.8397
UD	4818 (48%)	0	907	80.300	36	152.6016

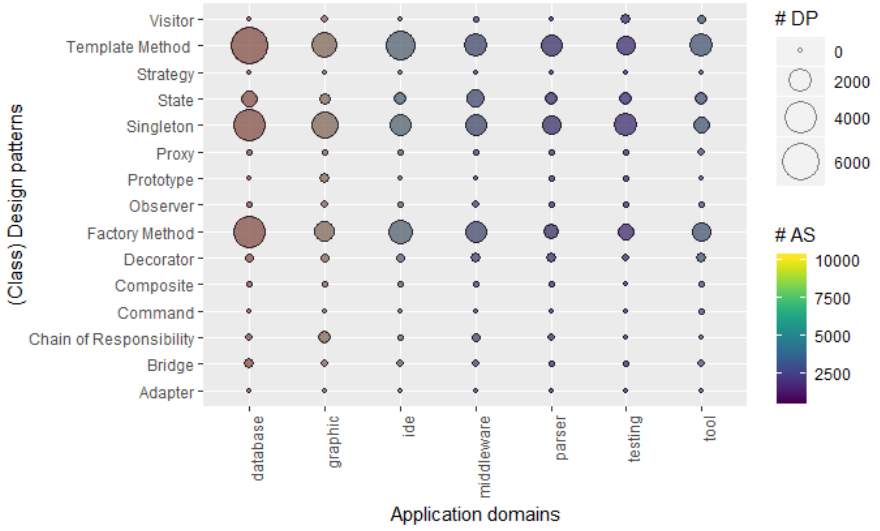


Figure 4.6: Frequency of AS and DP in 7 domains - Class level

Design pattern statistics Still regarding RQ1.1, Tables 4.6 present the collected statistics of the detected DP, respectively at class and package level, for each type. The most diffused DP are Template Method (18518 class dependencies, 7915 package dependencies), Singleton (13915 class dependencies, 7393 package dependencies) and Factory Method (11768 class dependencies, 8035 package dependencies). Adapter pattern was the only one to remain undetected during the analysis.

> In general, the most frequently detected patterns belong to the *creational* category, while the *structural* category contains the lowest number of patterns.

With regard to the applications domains, we did not find a specific prevailing DP.

Table 4.6: Dependency dataset - design pattern statistics

	# Dependencies (class)					*# Dependencies (package)*				
DP	**Tot.**	**%**	**Min**	**Max**	**Mean**	**Tot.**	**%**	**Min**	**Max**	**Mean**
Factory Method	11768	27%	0	2447	196.133	2245	37%	0	1741	133.917
Prototype	157	0.3%	0	126	2.617	25	0.40%	0	124	2.35
Singleton	13915	32%	0	2992	231.917	2372	39%	0	1394	123.217
Adapter	0	0%	0	0	0	0	0%	0	0	0
Command	20	0.04%	0	7	0.333	8	0.13%	0	3	0.133
Composite	94	0.2%	0	14	1.567	11	0.18%	0	6	0.45
Decorator	867	%2%	0	174	14.45	140	2%	0	74	6.683
Observer	133	0.3%	0	34	2.217	45	0.75%	0	10	0.917
State	3458	7%	0	551	57.633	1144	19%	0	438	37.85
Strategy	8	0.01%	0	4	0.133	3	0.05%	0	2	0.05
Bridge	421	0.9%	0	105	7.017	200	3%	0	61	4.517
Template Method	18518	42%	0	4449	308.633	2355	39%	0	1807	131.917
Visitor	319	0.7%	0	135	5.317	22	0.36%	0	68	2.017
Proxy	162	0.3%	0	22	2.7	32	0.53%	0	7	0.7
Chain of Responsibility	623	1%	0	356	10.383	33	0.55%	0	113	2.417

Figure 4.7: Frequency of AS and DP in 7 domains - Package level

RESULTS OF THE CORRELATION ANALYSIS To answer RQ2, we tested the correlation between AS and DP through the computation of Spearman *rho* and Kendall *tau* correlation coefficents.

Before running the test, we checked the normality of our variables with the Anderson-Darling test [172]. The null hypothesis is that the data are normally distributed; the alternative hypothesis is that the data are non-normal. We set the significance level at 0.05. We ran the test and rejected the null hypothesis for all the considered AS and DP variables, at both class and package level. We also generated Q–Q plots to confirm the results of the test, and they gave the same result. The scripts and the results of the normality tests can be found in the replication package.

The coefficent values are in the range $[-0.058, 0.134]$ for class dependencies, and $[-0.013, 0.117]$ for package dependencies.

As for the Spearman's analysis, the Kendall correlation was tested between all the possible pairs of AS and DP. The coefficent values are in the range $[-0.009, 0.019]$ for class dependencies, and $[0, 0.040]$ for package dependencies.

> Since all the coefficient values are very close to 0 (no correlation), we can conclude that this analysis did not discover an interesting relationship among AS and DP.

For this reason, we do not report the result tables, but they can be consulted in the replication package.

THE RESULTS OF MINING THE ASSOCIATION RULES In order to answer RQ2 and RQ3, we now present the results obtained from the association rule analysis, performed on both *class* and *package* dependency datasets. For each rule, we report their *antecedent* (Left Hand

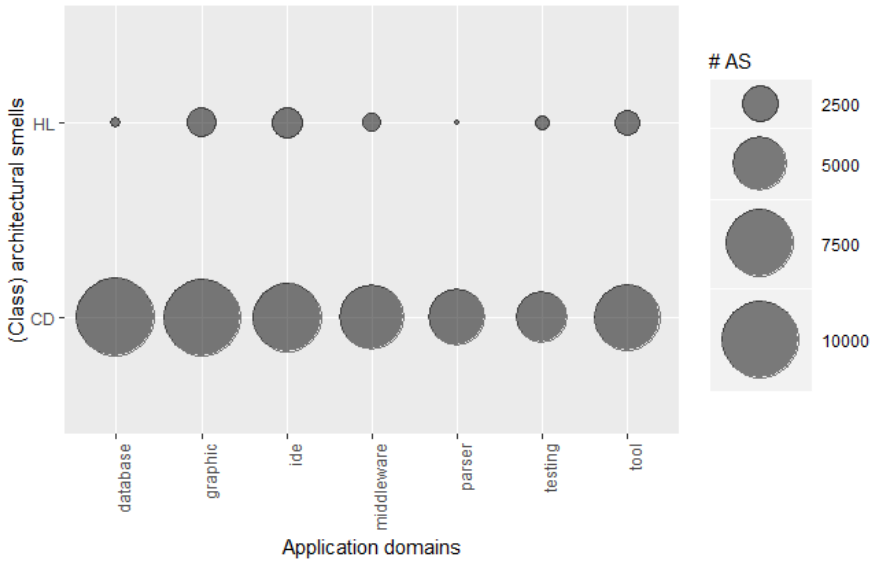


Figure 4.8: Frequency of AS in 7 domains - Class level

Table 4.7: Association rules at class level

LHS		RHS	Support	Confidence	Lift	Conviction
{HL, S}	→	{CD}	0.001	0.809	3.520	4.034
{CD, FM}	→	{TM}	0.004	0.628	7.676	2.473
{HL}	→	{CD}	0.024	0.661	2.879	2.276
{HL, TM}	→	{CD}	0.002	0.628	2.732	2.070

Side, LHS), their *consequent* (Right Hand Side, RHS) and commonly used metrics that evaluate its quality: support, confidence [4], conviction and lift [42].

We collected the rules with minimum support of 0.001 and confidence greater than 0.6.

Some of the rules contain only AS. This happens because we ran the rules extraction on all the dependency datasets, and some dependencies are affected at the same time by different smell types, but by no DP. For the sake of completeness, we leave such rules in our tables, even if we do not discuss them.

Class rules Table 4.7 reports 4 rules extracted from the (class) dependency dataset. Their support is in the range $[0.001, 0.024]$ and the conviction in the range $[2.070, 4.034]$. The involved AS are Hub-Like Dependency (HL) and Cyclic Dependency (CD), while among the DP there are Singleton (S), Factory Method (FM) and Template Method (TM). CD smell is the consequence of all the rules, apart from one where a TM pattern is present. This set is not surprising, since these are the most frequent types of AS and DP.

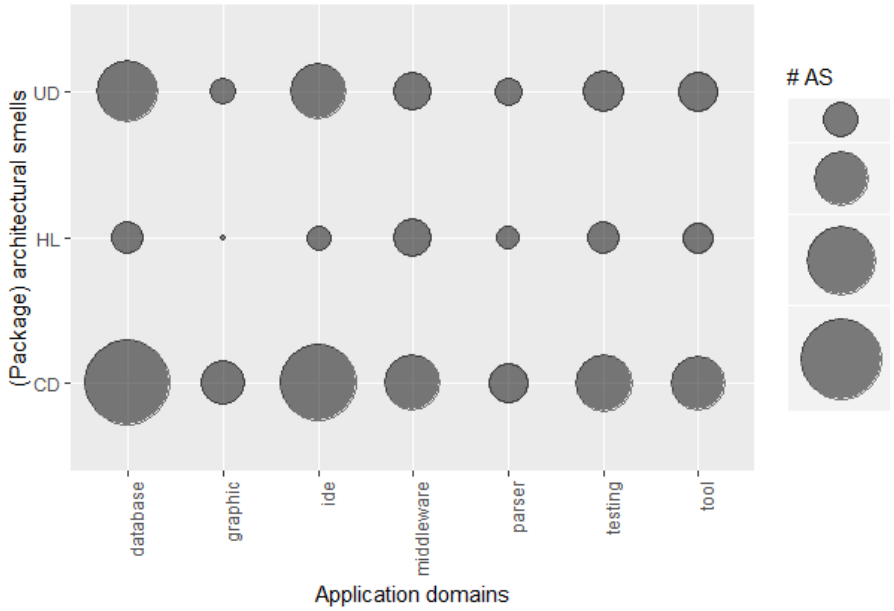


Figure 4.9: Frequency of AS in 7 domains - Package level

We conducted a manual validation of the 4 rules, by inspecting the code of the classes that match the rules. This was helpful for providing an interpretation of the discovered rules and to answer *RQ2* and *RQ3*. For each rule we provide *1) its description*, to understand how to read the rule, *2) a matching example*, found in the analysed projects, and *3) interpretation*.

We need to emphasize that the reported rules refer only to the subject dataset, and should not be freely extrapolated to other data. Additionally, they may not be applicable to all analyzed dependencies; in Table 4.7 column "Support" indicates the prevalence of the rule in the dataset and "Confidence" shows how frequently the consequent is collocated with the antecedent.

However, thanks to the association rule mining, we found that in some cases the presence of an AS is linked to the presence of specific DP. The aim of this interpretation is to provide practitioners and researchers with useful hints on what to do when specific combinations of AS and/or DP appear. For instance, we found some examples of false positive AS and cases of the potentially unsafe use of DP, e.g., when they are likely to introduce a new AS into the system.

In the following, we present an analysis of each class rule.

(R1)	`{HL, Singleton}` → `{CD}`

Description: if a dependency is affected by Hub-Like Dependency and Singleton, then this dependency also belongs to a cycle.

Example: from the manual validation of the projects from which such a rule was extracted, we identified two scenarios:

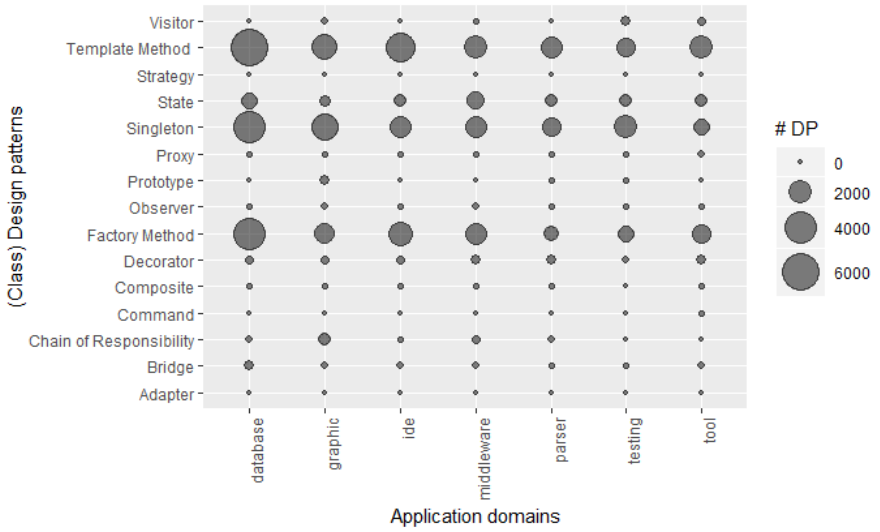


Figure 4.10: Frequency of DP in 7 domains - Class level

1. One of the two classes involved in the dependency is both an HL and a Singleton class.

2. One of the two classes involved in the dependency is a HL, while the other class is a Singleton.

This happens because our dataset considers dependencies at the expense of classes/packages, without the information about which of them is affected by a smell or a DP. With regard to *case 1*, we report the example of the Findbugs project. One of its packages named `edu.umd.cs.findbugs.gui2`, which groups the classes employed to build a graphical interface, contains a class named "MainFrame" which is a large HL (FanIn=96, FanOut=111) and a Singleton. This class has many Cyclic Dependencies with other classes from the GUI. The reason is that when the GUI frame creates a new listener (i.e., a new instance of a GUI class), it passes itself in the constructor to enable callbacks [248]. With regard to *case 2*, we report an example from the project antlr-3.4. `org.antlr.tool.Grammar`, which is an HL and depends on the Singleton class `org.antlr.misc.IntervalSet`. The dependencies between them form a cycle.

Interpretation: classes like "MainFrame" could be examples of false positives of the AS instances, where developers introduce Cyclic Dependencies in the project in order to implement a callback. Alternatively, this could be a case of improper implementation of the callback, which introduces a unique class which manages both the creation of the listeners and their callbacks. The fact that the class owns too much responsibility explains the presence of the Hub-Like Dependency smell, at the same time the presence of many Cyclic Dependency smells is caused by the callbacks.

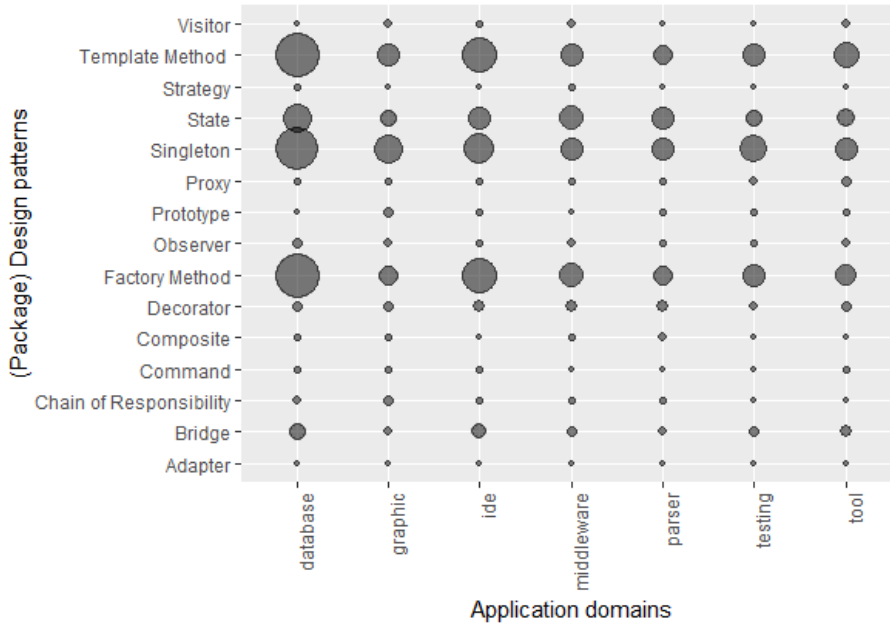


Figure 4.11: Frequency of DP in 7 domains - Package level

On the other hand, classes such as "Grammar" may be a symptom of the overlapping of a manager/core class (the hub) with the use of the Singleton class: it is neither a false positive, nor a bad implementation of the pattern; instead, it could be a particular *type* of Hub-Like Dependency. The fact that such dependencies are also implied in CD smells strengthens the idea that the hub is a collector of dependencies and centralizes the activity of the system. We found many classes matching this rule, hence this information could be helpful for refining the definition of AS and proposing a new classification for them.

(R2)	`{HL, Template Method}` $\rightarrow$ `{CD}`

Description: if a dependency is affected by Hub-Like Dependency and Template Method, then such a dependency also belongs to a cycle.

Example: in project Emma, a class `Attribute_info` in `com.vladium.jcd.cls.attribute` is both an HL and an AbstractClass of TemplateMethod. `SourceFileAttribute_info` extends `Attribute_info`, but the latter in some cases returns instances of the former (Broken Hierarchy [231]). `SourceFileAttribute_info` overrides the template method.

Interpretation: this is a false positive of Hub-Like Dependency, because dependencies toward the abstract class of Template Method affected by HL are actually resolved to its concrete classes. This rule is useful for refining the detection of the Hub-Like Dependency smell.

(R3)	{CD, Factory Method} → {Template Method}

Description: if a dependency is affected by Cyclic Dependency and Factory Method, then such a dependency also belongs to the Template Method DP.

Example: In the Derby project, belonging to the Database domain, many classes are *creators*, i.e., implement the Factory Method pattern. As an example, let us consider the class `Connection`, which is a core element in the management of a new database connection. This class is responsible for creating the class `Agent` and, at the same time, it forms a cycle with it. Moreover, it also implements the Template Method pattern, since other classes (e.g., `NetConnection`) inherit from this class and extend its template methods.

Interpretation: From the manual validation of the project matching the rule, it became apparent that classes involved in Factory Method are likely to be part of the same cycle. This is justified by the fact that newly created classes often need to use the creator class. Moreover, Factory Method classes tend to also implement the Template Method pattern. However, we did not find improper use of the two patterns. Hence, developers should only pay attention when using Factory Methods, because their use may lead to the introduction of new cycles.

(R4)	{HL} → {CD}

We do not provide an in-depth explanation of this rule, because it does not include a DP in its body, and because the relationship it describes has already been investigated for rule (R1) and (R2). In brief, we suggest that the relationship between HL and CD is justified by the fact that classes affected by HL are by definition involved in many dependencies, some of which can also be cyclic ones. Since the role of hub classes is usually central in the system, it is reasonable that other classes referred by hubs call back the hubs themselves. We find the discussion of the rules with DP more interesting, since when DP are combined with the presence of these two smells it gives us additional actionable information (e.g., about false positives).

Package rules Figure 4.12 shows how the conviction and support of the rules change depending on the *order* of the rules. We define the order of a rule as the number of distinct AS and DP appearing in the body of the rule [254].

As we can see from the plot, the rules with the highest conviction are the ones where order is equal to 2 and 3, hence we report all the rules with the order up to 3. The results of the package rule extraction are reported in Table 4.8. The Table shows the first 50 rules, ordered by conviction, with order $\leqslant 3$. The total number of rules is 188; they

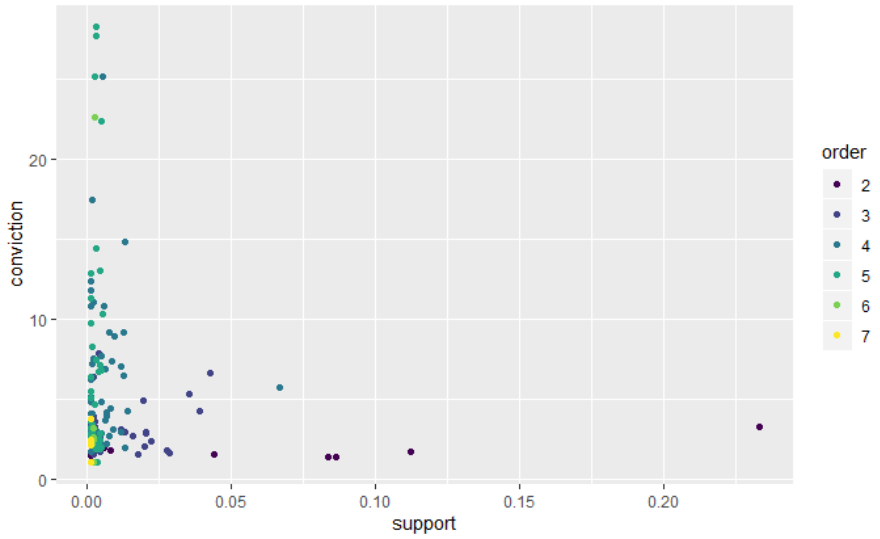


Figure 4.12: The order of package association rules

are documented in the replication package. Support is in the range $[0.001, 0.233]$ and confidence in the range $[0.610, 0.954]$.

We manually validated the rule with the highest conviction value, which puts the Visitor DP and the Cyclic Dependency AS into a relationship.

(R5)	`{V}` → `{CD}`

Description: if a dependency involves Visitor, then such a dependency is also involved in a Cyclic Dependency.

Example: The packages of the projects matching this rule appear to break the Visitor pattern in different packages, causing dependencies to spread from one package to the other, resulting in Cyclic Dependencies. An example is in the Eclipse project, where the packages `org.eclipse.core.internal.resources` and `org.eclipse.core.resources` contain the implementation of the pattern and are involved in a Cyclic Dependency. This is due to the anonymous class `ResourceTree$1` in package org.eclipse.core.internal.resources which implements the Visitor interface named `IResourceVisitor`, located in package `org.eclipse.core.resources`. The circular dependency appears because the package `org.eclipse.core.resources` contains some classes (example: `ResourcesPlugin` and `WorkspaceJob`) which depend on classes belonging to a different package named `org.eclipse.core.internal.resources`.

Interpretation: When the *concrete visitors* of the Visitor pattern are located in a different package than the Visitor interface, it is more likely that a Cyclic Dependency will occur between the two packages. This is due to the split of the elements of the DP into different packages. Developers should pay attention when they implement this pattern, in order to avoid the introduction of Cyclic Dependencies.

Table 4.8: Association rules at package level (top 50)

LHS		RHS	Support	Confidence	Lift	Conviction
{V}	→	{CD}	0.001	0.955	1.962	11.299
{P}	→	{CD}	0.001	0.840	1.727	3.210
{COR}	→	{CD}	0.001	0.636	1.308	1.412
{O}	→	{CD}	0.002	0.756	1.553	2.101
{D}	→	{CD}	0.006	0.729	1.498	1.892
{B}	→	{CD}	0.008	0.710	1.460	1.771
{ST}	→	{CD}	0.044	0.667	1.371	1.542
{S}	→	{CD}	0.086	0.630	1.296	1.389
{TM}	→	{CD}	0.083	0.614	1.262	1.331
{HL}	→	{CD}	0.112	0.700	1.438	1.710
{UD}	→	{CD}	0.233	0.840	1.726	3.205
{O,ST}	→	{CD}	0.001	0.800	1.645	2.568
{D,ST}	→	{CD}	0.002	0.655	1.346	1.487
{S,D}	→	{FM}	0.001	0.769	5.949	3.773
{FM,D}	→	{CD}	0.002	0.782	1.607	2.354
{S,D}	→	{CD}	0.001	0.846	1.740	3.338
{D,TM}	→	{CD}	0.002	0.870	1.788	3.938
{HL,D}	→	{CD}	0.002	0.846	1.740	3.338
{UD,D}	→	{CD}	0.002	0.919	1.889	6.334
{ST,B}	→	{FM}	0.003	0.636	4.921	2.394
{ST,B}	→	{TM}	0.003	0.610	4.500	2.219
{ST,B}	→	{CD}	0.003	0.753	1.549	2.081
{S,B}	→	{FM}	0.002	0.680	5.259	2.721
{FM,B}	→	{TM}	0.004	0.673	4.960	2.641
{B,TM}	→	{FM}	0.004	0.679	5.250	2.712
{FM,B}	→	{CD}	0.005	0.727	1.495	1.883
{S,B}	→	{TM}	0.002	0.700	5.161	2.881
{S,B}	→	{CD}	0.002	0.840	1.727	3.210
{B,TM}	→	{CD}	0.004	0.697	1.433	1.696
{HL,B}	→	{CD}	0.002	0.857	1.762	3.595
{UD,B}	→	{CD}	0.004	0.934	1.921	7.807
{FM,ST}	→	{CD}	0.018	0.661	1.359	1.514
{S,ST}	→	{CD}	0.011	0.833	1.712	3.069
{ST,TM}	→	{CD}	0.013	0.825	1.696	2.932
{HL,ST}	→	{CD}	0.016	0.808	1.662	2.680

Continued on next page

Table 4.8 – *Continued from previous page*

LHS		RHS	Support	Confidence	Lift	Conviction
{UD,ST}	→	{CD}	0.020	0.895	1.839	4.879
{FM,S}	→	{CD}	0.020	0.742	1.525	1.990
{FM,TM}	→	{CD}	0.028	0.676	1.389	1.584
{HL,FM}	→	{CD}	0.020	0.818	1.681	2.815
{UD,FM}	→	{CD}	0.035	0.904	1.858	5.333
{S,TM}	→	{CD}	0.027	0.712	1.464	1.783
{HL,S}	→	{CD}	0.020	0.823	1.692	2.899
{UD,S}	→	{CD}	0.042	0.922	1.896	6.619
{HL,TM}	→	{CD}	0.022	0.783	1.610	2.369
{UD,TM}	→	{CD}	0.039	0.879	1.808	4.252
{HL,UD}	→	{CD}	0.067	0.910	1.870	5.694

4.2.3 *Discussion*

In this section, we provide the answers to each RQ and discuss the obtained results.

RQ1: what is the distribution and prevalence of AS and DP in Java projects?

AS affect 24% of the class dependencies dataset and 57% of package dependencies. The most diffused type of AS is Cyclic Dependency, which affects 29.75% of the analyzed dependencies. *Megamek* is the project with the highest number of CDs, with 5504 occurrences. With regard to the Hub-Like Dependency smell, the most affected project is *DrJava*.

DP cover 19% of the dataset class dependencies and 34% of package dependencies: the most frequent DP is Template Method with 18518 instances at the class level (42%), and 2355 instances at the package level (39%). Hibernate is the project with the highest number of DP instances, in particular Template Method (4449 instances), Singleton (2992 instances), Factory Method (2447 instances) at class level; and Template Method (677 instances, %15), Singleton (680 instances, %15), Factory Method (609 instances, 13%) at package level.

We also counted the architecture dependencies which are involved at the same time in AS and DP (the number of examples where AS and DP are collocated). With respect to the total number of class dependencies which make up our dataset, this intersection represents 5%, and at package level it reaches 23%. This means that, at least at the package level, the collocation concerns almost 1/4 of the dependencies, which also makes it pertinent for qualitative investigation.

Within this question we also addressed the following topics:

RQ1.1: Is there a difference in the distribution of AS and DP with respect to the considered projects?

In general, both AS and DP grow in number as projects grow in size, where by size we mean the number of dependencies (Figures 4.2 and 4.3). Given that, we notice that the trend oscillates for both AS and DP. For instance, in correspondence with the interval of projects [jspwiki, hslqdb] (see the x-axis in Figure 4.2, on the right) the AS show a clear increase at the expense of the number of DP, while in slightly larger projects we observe a decrease of AS and an increase of DP. This effect occurs along all the graphics and could indicate that projects with a large number of DP have fewer AS.

However, this aspect goes over the aim of our research questions and should be investigated further in order to reach a clearer conclusion, for instance by increasing the number of analysed projects and by testing the correlation between the number of AS and DP of the projects.

RQ1.2: is there a difference in the distribution of AS and DP with respect to various application domains?

At the class level, the most AS-affected application domains are Database and IDE, specifically by Cyclic Dependency (which is also the most frequent smell in general) and Hub-Like Dependency. The same happens at the package level, where Database and IDE are strongly affected by Cyclic Dependency (Database from Unstable Dependency too), and Middleware domain is the most affected by Hub-Like Dependency.

In general, without considering the granularity, Graphics is the domain with the highest number of AS (in particular Cyclic Dependency), while the majority of the DP are present in the Database application domain. Hence, from the analysis conducted on our dataset, we can conclude that in general AS and Dps are equally frequent in the different domains, with a unique exception, the Database domain, where the number of DP far exceeds the number of AS.

RQ2: Which design pattern-architectural smell pairs display significant relationships?

As reported in RQ1, we found, especially at the package level, a share of dependencies where AS and DP are collocated. Hence, we analysed whether there is a relationship between specific types of AS and DP. In terms of a correlation coefficent, our analysis did not identify any significant relationships. The values of Spearman and Kendall coefficients are close to 0, indicating no correlation for any of the analyzed pairs of AS and DP. However, the results from mining the association rules are more interesting. In particular, we

extracted 4 rules regarding class dependencies that relate Hub-Like Dependency and Cyclic Dependency smells with Template Method, Factory Method and Singleton patterns.

We also identified some examples of false positive AS by manually validating these rules. We found that code suspected of the Cyclic Dependencies AS can be intentionally implemented inside a *callback*, which is similar to the Observer pattern [248]. With regard to the Hub-Like Dependency, we realized that when it is combined with Template Method, the smell is actually a false positive. This finding is particularly useful for refining the detection methods applicable to AS.

Moreover, we were able to extract 188 rules at the package level. We tried to provide an interpretation of the resulting rules by manually reviewing the code of the analyzed projects. We did this for all the rules at the class level and for one rule at the package level. By manual validation we identified cases where a DP led to the presence of a specific smell. For instance, from the analysis of a package rule *we discovered that the implementation of Visitor pattern that is spread across multiple packages is more likely to introduce Cyclic Dependencies among the packages*. Some patterns have also been found to be defect-prone in other studies: Observer and Singleton [250], Composite, Prototype, and Adapter-Command [25]. In particular Sousa et al. [226] found the Adapter-Command pattern to be highly correlated with code smells, which are usually indicated as the counterpart of AS at code level, i.e., symptoms of poor software quality. Although the results reported in the literature are not fully consistent, they indicate that some patterns appear more troublesome than others. Our results seem to partially confirm these findings and can be useful to developers, who should pay attention when implementing DP that could be associated to AS.

RQ3*: *Can the presence of architectural smells imply the absence of design patterns?* or vice-versa *Can the presence of design patterns imply the absence of architectural smells?

We counted the number of dependencies where only AS are present, without DP: at the class level they comprise 19% of the dataset, while at the package level 34%. On the other hand, dependencies involved only in DP are 14% at class level and 12% at package level. Given that the AS and DP collocation is 5% (class) and 23% (package), we could say that, on the basis of the numbers, there are more cases where the two concepts are mutually exclusive, i.e., more examples where the presence of one of the two excludes the other.

4.2.4 *Threats to Validity*

In this section, we discuss threats to the validity of our study, following the structure suggested by Yin [264].

Threats to **construct validity**, which concern the identification of the measures adopted, can occur due to errors in the data extraction and preparation phases. Moreover, we build and rely on a dataset based on object-oriented dependencies: there could be errors in the construction of the dataset, and we could have extracted biased results affected by the dependency representation. However, we relied on well known R libraries (e.g., `dyprl`) to manipulate our data and we manually checked our dataset and published both datasets and analysis results in the replication package[4]. Finally, some DP are related to methods whose granularity level was not considered in this study, and we aggregated class data to obtain a representation of DP at the package level. Hence, the analysis of the package-level DP and AS could have led to erroneous conclusions. On the other hand, we carefully explained our aggregation method and we provide examples of manual validation also for results at the package level.

Threats to **internal validity** are factors that could have affected the results obtained. In our case, they may be due to the choice of the statistical methods used for the analysis of the dependency dataset and their implementation in the used tools (R libraries and KNIME platform). We mitigate this threat by relying on multiple sources, such as similar empirical studies [78] conducted on code smells and DP correlations [254].

Threats to **external validity** refer to the generalization of the results beyond the original setting. They may arise from the nature of the projects used in our study. We analyzed only projects written in Java and that are publicly available. However, we partially mitigate such issues by analyzing a large number of projects (60). Another threat is related to the definition of software domains. Even though we relied on the categorization provided by the Qualitas Corpus [238], we decided to merge some of them in order create a more balanced dataset, which could have influenced our results.

Threats to **reliability** concern the correctness of the conclusions reached in our study. We rely on two tools (Arcan and Pattern4) to extract dependency information and detect AS and DP in the analyzed projects. Both tools could be subject to systematic bias in the detection. Such threats are partially mitigated by the provided replication package and the fact that both tools are available, validated and can be applied to any compiled Java project. Validation of Arcan results has been performed on ten open source projects [19] and on two industrial projects, with a high precision value of 100% in the results and 63% of recall [21]. Moreover, the results of Arcan were validated using the feedback provided by practitioners working on four industrial projects [155]. With regard to Pattern4, the tool has been validated on three open source projects [244]. The precision of all the examined patterns for all projects is 100%. Recall is 100% except for

4 https://drive.google.com/drive/folders/1ONSTAwyvK9d7gGp70kgLXfDZvkU80xz1

2 patterns: Factory Method (%66.7, %25 and %100, for the 3 projects) and State (%95.6, %91.6 and %100).

The differences in the recall between Arcan and Pattern4 may affect the conclusions reported in the answer to RQ1, where we compared the distribution of AS and DP in the dataset. However, to address the tool bias, we manually cross-validated the results, which also revealed other interesting insights that have not been found by the static analysis.

We need to acknowledge that, despite our efforts, our study does not provide definitive answers, especially to RQ2 and RQ3. The qualitative analysis of association rules (RQ2) was limited to a few examples, while the answer to RQ3 relies only on quantitative data. The latter threat is mitigated once again by the large number of analysed projects. Concerning our qualitative analysis, we reported the exact names of the classes/packages that we manually analysed, and since we considered only public projects, our statements can be easily verified.

4.2.5 *Final remarks*

We investigated whether the presence of AS in a project *influences* or *is influenced by* the presence of DP, under the hypothesis that the latter can sometimes have a negative impact on software quality [226] [120] [75] [258]. We studied the presence of AS and DP in 60 open source Java projects and explored the possible relationships that may occur among specific types of AS and DP. We built a dataset with the results obtained from the execution of two static analysis tools, Arcan (for AS detection) and Pattern4 (for DP identification). The dataset is *dependency oriented*, i.e., we associated information on AS and DP to object-oriented dependencies, since both AS and DP are commonly recurring *structures* of the software architecture. We collected statistical information about the frequency of AS and DP and their collocation: all the analyses were conducted separately at two granularity levels, class and package, and we also studied the results in relation to the application domains of the analyzed projects. Then, we performed a correlation analysis with two different coefficients, Spearman *rho* and Kendall *tau*, in order to detect possible statistical correlations among AS and DP. Finally, we mined our dependency dataset to extract association rules and, consequently, possible associations among AS and DP.

Our results show that in our dataset there are more examples of dependencies which are involved *only* in AS and *only* in DP, i.e., our data seems to confirm that they are mostly mutually exclusive concepts. However, from the qualitative analysis we performed with the association rules on the collocation examples, we found hints about what happens when AS and DP overlap.

There are indeed some connections between the co-occurrence of specific types of AS and DP. Some of them are effects of AS false positives instances, for instance the Template Method can be a signal of a Hub Like false positive. However, some relationships occur because specific implementations of DP can imply the introduction of AS, as happens for the Visitor pattern, which can cause the introduction of Cyclic Dependencies. Both results are useful in different aspects: by studying design constructs such as DP we can gain interesting ideas on how to enhance AS detection; being aware of the fact that the implementation of a DP can lead to effects contrary to intentions (i.e., the introduction of *bad* design decision), knowledge of AS-DP relationships helps developers to focus their attention on specific fragments and structures. Moreover, this indicates the need to develop new tools able to spot such smells, starting from the presence of specific DP.

4.3 ARCHITECTURAL SMELLS EVOLUTION AND CORRELATION: AN EMPIRICAL STUDY

This section describes an empirical study conducted on the evolution and correlation of six architectural smells[5]. The aim was to gather an insight about hidden relationships among different types of AS and provide useful takeaways concerning AS nature and causes. The study is guided by three Research Questions related to three main subjects: AS frequency, AS evolution, and AS relationships. We exploit our Arcan tool (see Section 2) for the detection of the AS.

We first analyzed the frequency of each type of AS in each analyzed project, with the aim to identify the smells which occur in larger number than others. Then we studied the evolution of the detected AS. In particular, we checked whether a trend is present or not in the data, to understand if smells (also depending on their type) tend to increase/decrease in number during the evolution of the software projects. Finally, we studied the relationships which may occur between different types of AS. This kind of study has been widely explored for code smells [22, 254, 261], but few works, according to our knowledge, have been done exclusively on AS. Knowing that two kinds of smells tend to occur together tells us more information about the nature of such smells and can be a hint for the definition of new categories of AS, being useful for researchers and developers of detection tools. The correlation of smells could be also the signal that there are common, recurring design problems behind different types of smells. Finally, the collocation could be a hint of the presence of false positive smells: in certain cases the common cause behind the collocation is the intention of the developer of building the component in a specific way, instead of being a design problem. We discuss some examples of false positives in Section 3, starting from the feedback we received in past works from industrial developers [155][79][217]. With this perspective, studies on correlation and collocation can help in a difficult task: to distinguish real problems from intended solutions. This becomes especially crucial for AS detection tools, which often indicate the presence of AS which are not considered problems by the developers (see Section 3).

We exploited well know statistical techniques to conduct the study, such as correlation analysis, Principal Component Analysis (PCA) and trend analysis. We ran Arcan on 10 Open Source Java projects, with about 10 versions each, for a total of 98 projects. In brief, the main contributions of this work are:

- Evolution analysis of architectural smells;
- Correlation and collocation analysis of architectural smells.

5 A publication has been extracted from this study, submitted to the Journal of Systems and Software

Researchers can benefit from our work since we provide empirical evidence of AS correlation, collocation and frequency, on the top of which future works of the field can refer to. Developers on the other hand can increase their knowledge about AS, e.g, learn which are the Open-Source projects that they exploit and are most affected by smells. We highlight such hints in the discussion. Additionally, we supply in the replication package[6] our dataset and the scripts used to conduct our analysis, which can be exploited for other future works.

4.3.1 *Architectural Smells Evolution and Correlations: Study Design*

As outlined above, we aim to investigate the possible relationships among different types of AS, over the projects' development history, and analyze also the evolution of the AS in order to answer the following Research Questions:

- *RQ1: which are the most frequent types architectural smells?* We report the number of AS detected in our dataset and identify the most frequent types; moreover, we highlight the most affected projects.
- *RQ2: do architectural smells follow a particular trend during the development history?* we study the evolution of smells over the projects' history, to understand whether there is a specific trend in the analysed data. We compare smell evolution with the project evolution, in terms of number of dependencies, i.e., the references among the different classes or packages, and the total number of Lines of Code.
- *RQ3: is there a relationship among the different types of AS?* We aim to discover which kinds of AS are correlated and collocated. For correlated smells, we mean two types of AS which show some kind of statistical relationship in the dataset. For collocated smells, we mean two or more types of smells which affect the same components [261].

MOTIVATION: The answers to *RQ1* and *RQ2* can be useful for developers to focus on specific types of smells despite others: they must pay particular attention to the ones which show a positive trend because it means that these smells tend to increase during project development; they also should take care of the ones which are most frequent, because such smells may appear in their systems as well.

The motivations behind *RQ3* are multiple. First, the discovery of AS correlations and collocations can be useful to researchers and tools' developers to better understand the AS nature. Moreover, the recur-

6 https://drive.google.com/drive/folders/1z0NEcy_e0xbNi6DVD9L2seW1Xp49scnA?usp=sharing

rent relationship among different types of AS can be the manifestation of deeper, hidden and recurrent design problems: the possible root causes of AS, which are more difficult to identify. The collocation of AS can also help developers and researchers in finding false positive instances of AS. Consider for instance the evidence of collocation among Hub-Like Dependency, God Component and Feature Concentration at package level. We found it thanks to two association rules, extracted from our dataset: {GC, HL} → {FC} and {GC, FC} → {HL}. Let's analyse one example of package which matches these rules: com.google.common.collect, a Guava package. In this case, the package:

- has 18 182 Lines of Code (LOC) → **GC**
- has balanced ingoing (Fan In) and outgoing (Fan Out) dependencies → **HL**
- addresses 22 features → **FC**

In practice, it means that this package is affected at the same time by all the three smells, as depicted in Figure 4.13: the large node in the middle represents the package, surrounded by other packages. The different textures inside the node present the multiple features addressed by the smells. These are the results from the analysis of Arcan; we then looked for the Guava code on Github. By checking the project documentation, it appears that this package contains *"generic collection interfaces and implementations, and other utilities for working with collections"* [94]. This is consistent with what we understand from the collocation of the smells: this package is very large, with a lot of incoming and outgoing dependencies, and addresses more concerns than every other package in Guava. The single smells are problems indeed, but together form a pattern which matches with the actual purpose of the package. A package full of utilities is intended to be used by many parts of the project, and by design implements more than one concern. We are in front of an example of false positive AS, or better, the single types of smell actually affect the package, but their collocation give us additional information about how to contextualize the package, i.e., hints about the possibility of being a utility package. This is one aspect that can be investigated thanks to the collocation analysis.

ANALYZED PROJECTS AND COLLECTED DATA We analyzed 10 projects, belonging to 3 different organizations (Apache, Eclipse and Google) and of different sizes. For each project, we took in consideration about 10 versions (we will use the term "version" to generically refer to the different releases), for a total of 98 projects. The time span for the analyzed versions ranges from 1 year to 5 years, and is comparable to other empirical studies on software evolution (e.g., 6 years,

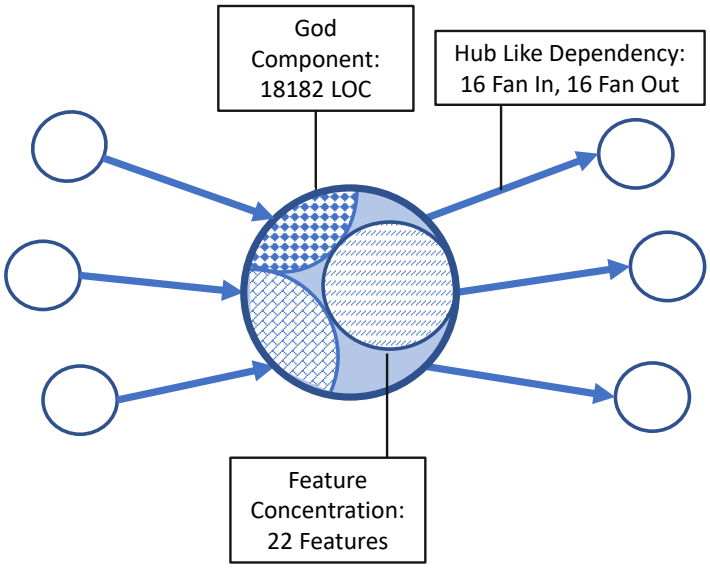


Figure 4.13: Example of collocation of three architectural smells - Guava

as reported in [193] and [181]). On average, the total number of Lines of Code (LOC) of the analyzed projects varies from 23 732 to 397 491. Table 4.9 reports some information about the projects: their name, the organization they belong to, the number of versions analysed in this study, the number of years occurred between the first and the last analysed version, the number of lines of code (LOC) both for the first and last analyzed version, and finally the number of classes (NOC) and packages (NOP) for the first and last analyzed version. We choose this set of projects because they are Open Source and publicly available on Github [93]. Moreover, they belong to different organisations and different application domains: in this way we ensure diversity in how architectures are designed and we collect data from varied sources. They have also been used in other empirical studies on code smells [97, 188, 261, 268].

Table 4.9: Detail of the analysed projects

Name	Org.	# Versions	Time span (years)	LOC (first)	LOC (last)	NOC (first)	NOC (last)	NOP (first)	NOP (last)
Jmeter-core	Apache	10	4	40 630	49 846	1575	1673	169	185
Mahout-mr	Apache	10	2	63 310	61 301	971	953	104	103
Maven	Apache	9	5	78 496	83 886	382	393	64	64
Struts	Apache	10	3	151 703	159 008	1679	1698	79	79
Collections-core	Eclipse	9	2	131 079	134 578	343	344	111	111
Jgit-core	Eclipse	10	1	105 009	108 432	1394	1404	56	57
Gson	Google	10	4	23 732	25 161	690	752	25	27
Guava	Google	10	2	339 228	397 491	762	773	23	23
Tink	Google	10	2	33 074	44 681	423	611	25	27
Truth-core	Google	10	2	28 740	33 421	115	121	4	4

For each project, we ran Arcan and collected data about the number of the six AS (CD, UD, HL, FC, SF, GC) in relation to the projects' architectural components (class and packages).

DATASET Starting from the raw results of Arcan, we build a dataset which we use for the analysis. Differently from the dependency dataset introduced in the previous work (see Section 4.2), in this study each observation corresponds to a single architectural component (class or package). The reason is that, differently from design patterns, the information about the type of dependency and its direction are not essential for the analysis of AS. Moreover, extracting from Arcan only the component information requires less computational resources, and given the quantity of analysed data (10 versions of 10 projects, for a total of 87429 data points), this choice allowed us to fasten the study process.

The set of features associated to a single observation is:

- *project*: the name of the project the architectural component belongs to;
- *versionIndex*: an index to keep track of the analysed version the observation refers to;
- *name*: the name of the architectural component;
- *componentType*: the type of component, class or package;
- *noSmell*: binary feature, true if the component is not affected by any smell;
- *AS1 ... AS#*: the number of smells affecting the component, one feature per type of AS. These features become binary in the binary version of the dataset.

Moreover, we created a variant of the original dataset by transforming AS features to binary, indicating only the presence/absence of smells on the architectural components. We exploit this variant to study the collocation of AS. In the next sections, we report the results of the analysis of the entire dataset, by project and by version.

ANALYSIS For all the analysis we exploited the R language [241]. In addition to computing the AS frequencies, we conducted 4 different statistical analysis, whose aim were to investigate the evolution and possible inter-smell relationships of the AS introduced in Section 2.2. We ran them on the results of Arcan, after its execution on the projects listed in Table 4.9.

- **Computation of AS frequencies**. In order to answer RQ1, we computed the frequencies of AS for each project, for each type of AS, at each granularity level.

- **Evolution analysis** In order to answer RQ2, we conducted trend analysis to understand how AS values evolve overtime. We exploited the *Mann-Kendall test*, which is a non-parametric test able to assess if there is a monotonic upward or downward trend of the variable of interest over time. The null hypothesis for this test is that there is no monotonic trend in the series. The alternate hypothesis is that a trend exists. This trend can be positive, negative, or non-null.

- **Correlation and collocation analysis** In order to answer RQ3 we ran a set of statistical analysis to investigate both correlation and collocation of AS. we ran the *Spearman correlation test* [227] to evaluate the correlation among different types of AS. We consider significant only tests with p-value < 0.05. This test is run on the number of smell instances, divided by type, collected from the entire dataset. We chose Spearman since it does not require the input data to be normal: we checked the normality of our variables with the Shapiro-Wilk test [219] and found that they were not normal. To investigate collocation, we exploited three different techniques. 1) First we ran the *pairwise correlations* on the binary variant of our dataset. Usually, the ϕ coefficient is used to obtain the pairwise correlation between two dichotomous variables, but here for simplicity we ran Pearson correlation test on all the AS variables, since this method is equivalent to the *phi* coefficient when ran on binary values [61]. We consider significant only tests with p-value < 0.05. 2) We also run *Principal Component Analysis (PCA)* in order to identify the collocation of AS. 3) Finally, we exploit *association rules mining* to identify relationships among different types of AS. We use an implementation of the Apriori algorithm [4], with the *minimum support* set to 0.001 and *confidence* set to $\geqslant 0.6$, as used in other empirical studies [253]. We also discard redundant rules [23], i.e., rules which are equally or less predictive than a more general rule and have the same items on the consequent side, but one or more items less in the antecedent side.

The following sections present the results of the introduced analysis.

4.3.2 *Results*

We illustrate here the results of all the analysis and the answers to the RQs.

RESULTS FOR RQ1 We show in Table 4.10 and 4.11 the number of detected AS, for each analysed project, namely at class and package level, in order to answer RQ1. The table reports the smell counts for

both the first and the last versions of the given project, divided depending on the type of the affected component (class or package).

The type of AS which is most common in the dataset is Cyclic Dependency (CD) at class level, followed by CD at package level and Scattered Functionality (SF). This result is consistent with the results of previous works [19][17][221], where CD (at both levels) usually counts the highest number of occurrences in the projects under analysis. However, Guava and Truth projects are not affected by CD at package level, in any of the analysed versions: both of them belong to the Google organisation. Since the complete absence of CD is atypical, we compared the two projects' characteristics to find possible commonalities which could explain why their CD frequencies are different from the other projects. We found out that both projects have a small number of packages despite the other projects, while having a total number of lines of code (LOC) below (Truth) and over (Guava) the average. If we compute the ratio of LOC on the number of packages for both projects, we find out that it is higher than the other ones. This means that packages in Guava and Truth are larger than in the other projects. We manually checked their code and we found out that their package structure (containment tree, see Section 2.1) is not nested, but all classes belong to packages at the same level (the first) of the containment tree. Thus, there are not small, nested packages containing few classes, but only few large packages. This specific way to organize packages might be the reason behind the lack of CD at package level in these two Google projects. We aim in future works to further study the characteristics that are common to both projects, to identify a kind of "recipe" for CD-free software projects.

In the same way, Mahout and Maven do not have God Component (GC) instances. Finally, Tink and Truth have no Scattered Functionality (SF). In general, the peculiar projects are the ones from Google, namely Gson, Guava, Tink and Truth. They have the smallest amount of AS (computed as the sum of all types of smells) at package level, ranging in $[6, 23]$ in the first versions and in $[6, 57]$ in the last versions. At class level this fact is less apparent, even if the project with the smallest number of AS is still part of the Google organisation (Tink). While the projects having the largest number of AS, both at class and package level, are Jgit and Jmeter: they are particularly affected by Cyclic Dependency.

RQ1: which are the most frequent types of architectural smells? The most frequent architectural smell is Cyclic Dependency, at both class and package level. The most affected projects (by all types of AS) are Jgit and Jmeter. Moreover, the considered Google projects are all designed in a similar way and the resulting architectures are affected by few or even zero Cyclic Dependency,

Table 4.10: Number of architectural smells - Class level

	First version			*Last version*		
Project	**CD**	**HL**	**Tot.**	**CD**	**HL**	**Tot.**
Collections-core	299	5	304	300	5	305
Gson	453	12	465	620	2	622
Guava	558	2	560	609	2	611
Jgit-core	6351	4	6355	15675	38	15713
Jmeter-core	3366	18	3384	5994	18	6012
Mahout-mr	306	4	310	302	4	306
Maven	148	15	163	159	22	181
Struts	528	21	549	542	17	559
Tink	118	14	132	154	14	168
Truth-core	141	6	147	175	5	180
Total	12268	101	12369	24530	127	24657

Table 4.11: Number of architectural smells - Package level

	First version							*Last version*						
Project	**CD**	**GC**	**HL**	**UD**	**FC**	**SF**	**Tot.**	**CD**	**GC**	**HL**	**UD**	**FC**	**SF**	**Tot.**
Collections-core	37	1	1	5	2	135	181	39	1	1	6	0	0	47
Gson	11	1	2	3	2	2	21	13	1	1	3	1	2	21
Guava	0	4	1	1	1	2	9	0	4	1	1	0	0	6
Jgit-core	153	7	2	9	3	34	208	293	6	7	11	0	0	317
Jmeter-core	483	1	6	20	1	2	513	490	1	7	22	0	0	520
Mahout-mr	30	0	8	15	4	40	97	30	0	10	14	0	0	54
Maven	24	0	4	12	7	39	86	53	0	1	11	0	0	65
Struts	60	3	2	12	4	62	143	85	4	10	11	0	0	110
Tink	15	1	1	3	2	0	22	44	2	2	3	0	0	51
Truth-core	0	1	4	0	1	0	6	0	1	4	0	0	0	5
Total	813	19	31	80	27	316	1286	1047	20	44	82	1	2	1196

indicating a possible hint about how to design CD-free software systems.

RESULTS FOR RQ2 In order to answer RQ2 we ran the Mann-Kendall test to investigate whether AS have a trend over the development history. All the AS show a significant positive trend (increasing) except for UD smell in project Mahout. Also AS at class level show positive trends, but only for one type of smell: CD. The projects where the trend is significant are 6, namely Gson, Jmeter, Mahout, Struts, Tink and Truth. Thus, in general, all AS tend to increase overtime.

Since a set of the smells, namely HL, UD and CD are based on dependency issues, we also ran the Mann-Kendall test to check the trend of package and class dependencies over time. The Total Number of Dependencies metric is computed for each component (class or package) belonging to a project and consists in the sum of the number of ingoing dependencies (from other components) and the number of outgoing dependencies (vice-versa). Our goal was to verify if there is a relationship between the increase of the smells and the trend of dependencies. The results at package level showed that the dependency trend is positive for the following projects: Jmeter, Mahout, Struts and Tink. Except for Jmeter, all these projects showed also positive trends for HL, CD and UD. Thus, it is likely that a project whose package dependencies tend to increase will be affected by more smells during its evolution. However, at class level, only Jgit, Mahout and Guava showed an increasing dependency trend; among them, only Mahout showed an increasing trend of CD at class level. Additionally, we checked the correlation between the number of AS and the total number of Lines of Code (LOC) of the projects, during their evolution (see Table 4.12 and 4.13, notice that the bold values are the significant ones). For smells at package level, we found significant Spearman correlation between LOC and Cyclic Dependency($\rho = 0.659$) and God Component ($\rho = 0.715$). For smells at class level, we found a significant relationship between LOC and Cyclic Dependency ($\rho = 0.860$).

Such results are a confirmation of the outcome we obtained in other works [80][196],i.e., projects size has a relationship with the increase/decrease of smells, but it is not the only variable affecting the trend. For instance, the frequent application of refactoring activities has proven effective in managing AS [231] and more in general Architectural Technical Debt [240].

RQ2: do the presence of AS follow a particular trend during development history? Given our results, we can state that in the case of the 10 analysed project, architectural smells tend to grow in number as the project development proceeds i.e. they show a positive trend. Moreover, by analysing the evolution of size in terms of number of dependencies and number of lines of code,

Table 4.12: AS and LOC correlation - package

	CD	GC	HL	UD	FC	SF	LOC
CD	1.000	0.133	**0.402**	**0.788**	**0.220**	**0.407**	**0.659**
GC	0.133	1.000	-0.183	**-0.325**	**-0.288**	-0.194	**0.715**
HL	**0.402**	-0.183	1.000	**0.515**	**0.225**	0.053	0.030
UD	**0.788**	**-0.325**	**0.515**	1.000	**0.434**	**0.550**	**0.324**
FC	**0.220**	**-0.288**	**0.225**	**0.434**	1.000	**0.659**	-0.112
SF	**0.407**	-0.194	0.053	**0.550**	**0.659**	1.000	0.179
LOC	**0.659**	**0.715**	0.030	**0.324**	-0.112	0.179	1.000

Table 4.13: AS and LOC correlation - class

	CD	HL	LOC
CD	1.000	-0.169	**0.861**
HL	-0.169	1.000	0.074
LOC	**0.861**	0.074	1.000

it appears that the evolution of CD at both class and package level is the most related to the evolution of size, in the analysed projects. This results tells us that size is one of the factor contributing to the growth of AS, however it is not the only one.

RESULTS FOR RQ3 We report the results for each analysis conducted to answer the RQ3 and summarize our findings at the end of the section.

Spearman correlation test results As part of the answer to RQ3, we ran the Spearman correlation test on the data relative to the different types of AS (Table 4.14 and 4.15, bold values are significant ones). We consider correlated pairs of smells with at least $|\rho| \geqslant 0.3$ [56]. Differently from the collocation analysis, we do not exploit the correlation test to study smells which co-affect the same architectural component, but we compare the total number of smell instances, divided by type, detected in the entire dataset. The smell pair showing the highest positive correlation at package level is Cyclic Dependency and Unstable Dependency ($\rho = 0.763$), followed by Hub-Like Dependency and Unstable Dependency ($\rho = 0.523$). Also Hub-Like Dependency and Cyclic Dependency showed a positive correlation, but very close to 0 ($\rho = 0.302$).

Concerning negative correlation, we found some examples, e.g, for GC and SF, but with values very close to 0. We also computed the correlation between Cyclic Dependency and Hub-Like Dependency

at class level: the resulting coefficient was positive, with value equal to 0.320.

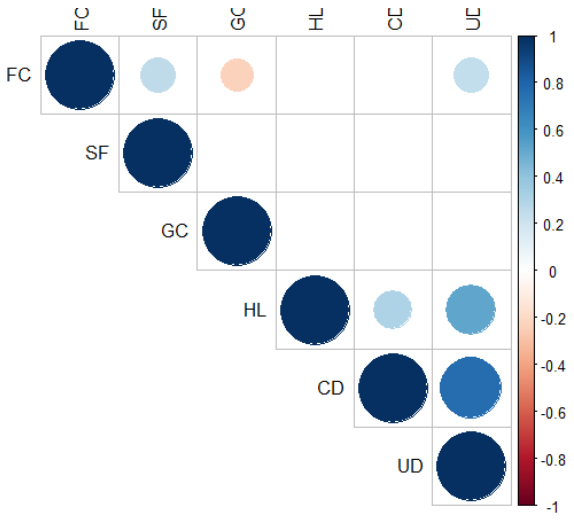


Figure 4.14: Spearman correlation coefficients - Architectural smells

Table 4.14: Spearman correlation test - package

	CD	GC	HL	UD	FC	SF
CD	1.000	0.050	**0.302**	**0.763**	-0.193	-0.153
GC	0.050	1.000	0.034	-0.166	**-0.220**	-0.074
HL	**0.302**	0.034	1.000	**0.523**	0.167	-0.046
UD	**0.763**	-0.166	**0.523**	1.000	**0.247**	0.094
FC	-0.193	**-0.220**	0.167	**0.247**	1.000	**0.259**
SF	-0.153	-0.074	-0.046	0.094	**0.259**	1.000

Table 4.15: Spearman correlation - class

	CD	HL
CD	1	**0.230**
HL	**0.230**	1

Pairwise correlation results To answer RQ3, we also investigated AS collocation. In particular, the pairwise correlation is the first analysis we did. We ran the computation of the Pearson coefficients on both dataset, class and package.

A blank cell is present if the corresponding smell pair had no significant correlation. We consider as significant tests with p-value < 0.05, while we consider correlated pairs of smells with at least $|\rho| \geqslant 0.3$ [56]. We did not obtain interesting results for smells at class level, and even

if we had 20 significant correlations at package level, the only pairs of smells whose values are far from 0 are formed by Cyclic Dependency and Unstable Dependency, with $\rho = 0.520$, Hub-Like Dependency and Cyclic Dependency, with $\rho = 0.303$.

Table 4.16: Pearson test - Architectural smells

AS	Rho	P-value
package		
CD, UD	**0.520**	**0E+00**
UD,HL	0.166	0
FC,HL	0.241	0
GC,HL	0.110	0
FC,GC	0.161	0
FC,UD	0.157	0
FC,CD	0.192	0
GC,UD	0.184	0
GC,CD	0.141	0
HL,CD	**0.303**	**0**
SF,CD	0.069	5E-08
SF,UD	0.078	6E-10
SF,HL	-	-
SF,FC	0.164	0
SF,GC	0.046	3E-04
class		
(CD,HL)	0.091025	0

Principal Component Analysis results The second analysis ran to answer RQ3 on AS collocation is the Principal Component Analysis (PCA). Before running the analysis, we verified that KMO of package dataset is above the threshold (0.562) and its sphericity is also satisfactory (Bartlett's test is significant, p-value equals to 0). Also the class dataset satisfacted KMO (0.5) and Bartlett's (p-value equals to 0). Therefore, both the datasets are suitable for applying PCA.

Figure 4.15 resumes the results of the PCA on packages. Correlated smells are located in the same sector of the graphics: Unstable Dependency and Cyclic Dependency are highly correlated; also Feature Concentration, Hub-Like Dependency and God Component are correlated; Scattered Functionality is not correlated with the other ones. Concerning classes, the PCA did not find correlation between Cyclic Dependency and Hub-Like Dependency.

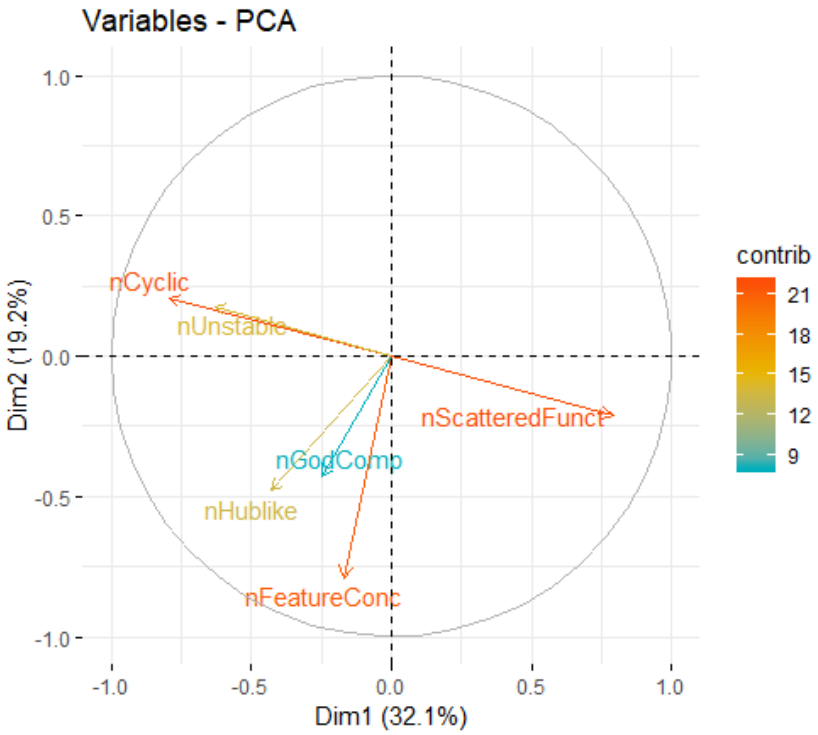


Figure 4.15: PCA results on package dataset.

Association rules analysis results Still concerning AS collocation, we now present the results obtained from the association rule analysis. For each rule, we report their *antecedent* (Left Hand Side, LHS), their *consequent* (Right Hand Side, RHS) and commonly used metrics that evaluate its quality: support (*Supp*), confidence (*Conf*) [4], conviction (*Conv*) and lift (*Lift*) [42].

We now report the collected metrics both for package and class dataset. We extracted many package rules (43) but here we report a selection of rules (20), based on their *order*. The order of a rule is the number of distinct AS appearing in the body of the rule [254]. We plotted the conviction and support of the rules change depending on their order (see in the replication package). We acknowledged that the rules of orders 2 and 3 appear more relevant (in terms of support) than other rules. Instead, we mined only one rule concerning class AS.

Concerning the package dataset, we found rules with support in the range $[0.002, 0.098]$ and conviction in the range $[1.988, 7.635]$ (excluded the rule with Confidence equals to 1 which entails infinite Conviction). Table 4.17 reports all the mined rules along with the quality metrics. All the types of smells appear in the rules.

We briefly comment the most interesting rules. The rule which shows the higher support is {UD} $\rightarrow$ {CD}, moreover 5 out of the 9 rules contains the UD smell in the antecedent (column *LHS*), which means that **CD and UD** are often collocated. Rules {GC, FC} $\rightarrow$ {HL}

Table 4.17: Association rules - Architectural smells

LHS		RHS	Support	Confidence	Lift	Conviction
Package rules						
{UD,FC}	→	{CD}	0.014	0.897	4.214	7.635
{GC,UD}	→	{CD}	0.013	0.888	4.170	7.006
{HL,UD}	→	{CD}	0.018	0.847	3.981	5.156
{GC,HL}	→	{FC}	0.005	0.769	19.170	4.159
{CD,GC}	→	{UD}	0.013	0.782	6.134	4.005
{HL,SF}	→	{CD}	0.015	0.798	3.751	3.903
{GC,FC}	→	{HL}	0.005	0.732	12.205	3.504
{UD,SF}	→	{CD}	0.036	0.774	3.635	3.478
{UD}	→	{CD}	0.098	0.770	3.617	3.421
{HL,FC}	→	{CD}	0.010	0.744	3.496	3.077
{UD,FC}	→	{SF}	0.012	0.763	2.753	3.049
{HL}	→	{CD}	0.042	0.704	3.306	2.657
{FC}	→	{SF}	0.026	0.636	2.297	1.988
Class rules						
{HL}	→	{CD}	0.008	0.724	2.217	2.446

and {HL, GC} → {FC} comprise the same set of smells, **GC, FC and HL**. The first rule states "If a package is large and with a low cohesion in terms of features, then it is more likely that it is also a hub of the system" which is reasonable, since the high number of dependencies which characterises HL smells could be caused by the fact that the package addresses too many responsibilities and thus is referenced by many other parts of the system. Instead, the second rule shifts FC smell with HL smell: this confirms the tendency of these smells of appearing together. We already discussed how this can be considered a false positive case of smell detection in Section 4.3.1. The intuition is that a package with such smells may be full of utility classes, used by all the other components of the system. Finally, with a lower Conviction, **SF** appears to be collocated with **UD** and **FC with SF**.

Concerning the class dataset, only one rule was extracted and is reported in Table 4.17. The rule confirms the collocation of HL and CD, with a good confidence (about 70%) and a positive conviction.

Table 4.18: Summary of correlation and collocation results

	Correlation	*Collocation*	
AS	**Spearman**	**PCA**	**Association r.**
	Package		
CD, UD	**+**	**+**	**+**
UD,HL	+		+
FC,HL		+	+
GC,HL		+	+
FC,GC	**-**	**+**	**+**
FC,UD	+		+
FC,CD			+
GC,UD			+
GC,CD			+
HL,CD	+		+
SF,CD			+
SF,UD			+
SF,HL			+
SF,FC	+		+
SF,GC			
	Classes		
HL,CD			+

4.3.3 *Final remarks on correlation and collocation results*

We resume the results of the 3 analysis we conducted in Table 4.18. We do not include the pairwise correlation results since they are not relevant. Here we gather the results of both correlation and collocation analysis: our aim is to highlight the relationship between *pairs* of different types of AS, from different points of view. Each row of the Table represents the relationship between two different AS (first column). For each conducted analysis, reported in the remaining columns, the significant results regarding the positive correlation of two smells are marked with a "+", the negative with "-". We highlight the Table rows corresponding to the pairs of AS which resulted in a relationship for more than two analysis.

One of the smells' pair which resulted in relation for all of the tests is **(Cyclic Dependency, Unstable Dependency)**. This show us that CD has a strong relationship with UD: they are both smells which regard the dependency structure of the system, and it is likely that

the presence of one of them, due to an incorrect dependency management, can lead to the introduction of the other. The relationship shows also a strong effect size in terms of both correlation and collocation, with Pearson ρ equals to 0.520 and Spearman ρ equals to 0.763. The other pair, (**Feature Concentration**, **God Component**), shows a negative correlation for Spearman, and positive for the other two tests: however, the Spearman test returned a value very close to zero (0.19), making difficult to support potential insights concerning the negative correlation.

Concerning the couples whose relationship was confirmed by at least two tests, the relationship between Scattered Functionality and Feature Concentration could be explained by the fact that if the concerns of a project are scattered among different packages, then it is more likely that those packages suffer of Feature Concentration, i.e., they implement too many (scattered) concerns. However, even if they appear to be both correlated and collocated, their effect size is small, compared to the other pairs of smells. The relationship between God Component and Hub-Like Dependency, as for the one between Feature Concentration and Hub-Like Dependency, could be explained by the example which we already discussed in Section 4.3.1, i.e., the three smells can overlap on the same component and actually disclose smells' false positive instances.

Concerning smells at class level, which were of two types, Cyclic Dependency and Hub-Like Dependency, two over the three analysis gave evidence of their relationship. Concerning correlation, Spearman coefficient resulted positive. Concerning collocation, only the association rule had positive conviction. However, these tests reported also small effect size. This means that a relationship is present, but it is not strong.

> ***RQ3: is there a relationship among the different types of AS?*** Our results showed that there are pairs of AS which are correlated and also collocated. The strongest ones (validated by all the conducted tests) are between Unstable Dependency (UD) and Cyclic Dependency (CD).

4.3.4 *Discussion*

Concerning the collocation results, we found relationships among more than two AS. By putting together the results of all the analysis, and in particular thanks to PCA, we found that, at package level, Cyclic Dependency and Unstable Dependency participate in the same component, with also a strong correlation. Instead, God Component, Hub-Like Dependency and Feature concentration belong to another component. Additionally, the extracted association rules put in relationship multiple smells: the most interesting is the one among

Hub-Like Dependency, God Component and Feature Concentration, which we suggest, as explained in Section 4.3.1, being an example of false positive AS detection.

The AS involved into the two *clusters* (FC, HL, GC) and (UD, CD) impact different design principles [27] and manifest themselves in different ways in the affected architecture. However, all clusters can be associated to macro-problems: FC, HL, GC affect one component at a time and have a relation with how the responsibilities of a system are assigned; UD and CD affect the dependency structure of the system and their instability. Our conclusion is that the AS in each cluster share a common cause which can explain the appearance of multiple smells in the same parts of the architecture. This could mean that resolving one single problem (the root cause) could lead to the resolution of all the AS. In Section 4.3.1 we discussed the hypothesis concerning the cluster composed by (FC, HL, GC), suggesting a scenario where the collocation of all the three smells actually represent a case of false positive smells, and the component is designed in such way *on purpose*. However, we propose an additional (and straightforward) scenario for such collocation: as long as additional and diverse features are added to a component (FC), the component grows in size (GC). At the same time, a large component (GC), which has a lot of different responsibilities (FC), could need to use (and be used) by many other parts of the system, resulting in a HL. Such interpretation of the collocation tells us that it is difficult to establish the root causes behind the presence of smells and thus it is also difficult to develop an automatic approach for the detection of such causes. The challenge is that the causes could be related to non-technical aspects hardly detectable by code analysis, e.g., the experience and choices of the development team. Thus, researchers developing this kind of approaches should take into consideration different data sources than code and develop models to represent both the AS and the *context* in which the AS are detected.

Observations: Sharma et al. [221] conducted an empirical analysis on the correlation between design and architecture smells: we now briefly discuss our work in relation to their results. Their distinction between design and architecture smells can be mapped to our distinction in class and package AS. *Concerning AS frequency*, we found out that the most frequent smell in the considered Java projects is Cyclic Dependency, at both class and package level. In the same way, Sharma found that Cyclic Dependency at both design and architecture granularities occur most frequently in open-source C# repositories compared to other types of smells. This further confirms its importance: developers should avoid to introduce this kind of smells in both the programming languages and also suggests that AS behaviour could be cross-language. This would mean that knowing how to deal with

certain types of smells in one language could be transferred to other (Object-Oriented) languages.

We also computed the Spearman correlation between the number of smells and the LOC of the considered projects, and found out that Cyclic Dependency at both granularity levels increases along with the size of the projects. Instead, Sharma computed the smell density (the average number of smells identified per thousand lines of code) of their projects and ran the Spearman correlation between density and LOC: their results indicate that the size of a project has no impact on the smell density of the project. They did not distinguish the type of smells when running the correlation and did not aggregate the results depending on the project version (they considered the total number of smells found in a given project *repository*) thus this can explain the difference in the two results.

Concerning AS correlation, Sharma tried to understand if certain smells at design level could be superfluous, i.e., indicate the same problem, respect to their counterpart at architectural level, and vice-versa: they analysed pairs of smells and found varying degrees of correlation. They interpreted this result as evidence that each type of smell provides value adding information. We computed correlation on different smell types at the same granularity level, thus our conclusions are not directly comparable with theirs. However, apart from few examples of medium-high smell correlation, we also did not find particularly strong correlations.

Also for what concerns *AS collocation*, we cannot directly compare our results. Sharma found out that few pairs of design and AS are collocated and this enforced the idea that the two concepts are separated. Thus they suggested that, even if architecture smells arise from code and implementation choices, there must be some additional (unknown) factors which cause their introduction. Similarly, we came to the conclusion that some types of smells may share a common cause: we found clusters of smells which impact on the same design principles and show some level of collocation in different analysis (PCA, association rules). Differently from Sharma, we also propose that this kind of analysis helps in identifying false positive AS: we discussed this aspect in Section 4.3.1.

4.3.5 *Threats to Validity*

We now discuss the threats to validity, following the structure suggested by Yin [264].

Threats to **construct validity**, which concern the identification of the measures adopted, can occur due to errors in the data manipulation and preparation phases. However, we relied on well known and widely used data manipulation libraries of the R language, and published all the scripts used during data preparation and analysis in the

replication package[7]. Threats to **internal validity** are factors which could have affected the results obtained. In our case, they may be due to the choice of the statistical methods used for the analysis and their implementation in the R libraries. We mitigate this threat by relying on multiple sources, such as similar empirical studies [78] conducted on code smells collocation [254] [261]. Threats to **external validity** refer to the generalization of the results beyond the original setting. They may arise from the nature of projects used in our study. We analyzed only projects written in Java and publicly available. Moreover, also the size of the projects, in terms of LOC, number of classes and packages could have influenced our results. For instance, the number of CD is correlated to the size of the project (as shown in the answer to RQ2, Section 4.3.2), and this fact could threat the validity of the answer to RQ1. However, we partially mitigate such issues by analyzing a large number of projects: 10 projects, about 10 versions each for a total of 98 single-version projects, coming from three different organizations and of different sizes (see Table 4.9). Threats to **reliability** concern the correctness of the conclusions reached in our study. We rely on Arcan tool to extract AS data from the chosen projects. The tool could be subjected to a systematic bias in the detection. Such threat is partially mitigated by the provided replication package and the fact that the tool is available, validated and can be applied to any Java project. A validation of Arcan results has been performed on ten open source projects [19] and on two industrial projects with a high precision value of 100% in the results and 63% of recall [21]. Moreover, the results of Arcan were validated also in industrial settings (see Section 2.2). Finally, another threat regards the possibility of replicating this study. To mitigate this point, we provide a replication package containing our dataset and the scripts used to built it. Moreover, each analyzed project is available since they are Open Source and available on Github [93].

4.3.6 *Final remarks*

In this study, we investigated AS correlation, collocation and evolution, through statistical analysis of multiple versions of 10 Open Source projects and discussed how our results relate with the ones obtained by other authors [221], confirming some of them.

We found out that the most present AS in the dataset is Cyclic Dependency, at both class and package level, and this confirms the results obtained by other authors in the field [221]. This smell has also the highest correlation with the evolution of the projects' size: developers should particularly pay attention to this kind of AS. Moreover, we analysed the correlation among the different types of AS and

7 https://drive.google.com/drive/folders/1z0NEcy_e0xbNi6DVD9L2seW1Xp49scnA?usp=sharing

found that they divides into two clusters, probably the results of the same cause underneath.

4.4 SUMMARY OF THE FINDINGS

We end the chapter by resuming the most interesting findings. We investigated the correlation of AS with design patterns, and also among the different types of AS themselves. Our aim was finding a relationship able to tell us something about the nature of AS: whether they cause the degradation of DP, or the contrary, if degraded DP cause AS; whether two or more smells tend to be collocated, meaning that they share some commonalities and may have the same root cause. All these kinds of information are useful for two reasons: enhancing the tools for AS detection, like Arcan, and provide guidelines to developers for managing AS.

For instance, we got useful hints from the association rules involving AS and DP: the specific combination of some smells, patterns and application domain indicate that the smell is a false positive. We report the case of Cyclic Dependency, collocated with Hub-Like Dependency and Singleton in the projects belonging to the graphical domain, signaling the presence of a *callback* ($\{HL, S\} \rightarrow \{CD\}$). This specific example is interesting, because the association rule that models it has a strong confidence (the relationship between the antecedent and consequent is strong). The rule could be embedded in a detector, and help filtering out the CD false positive instances. Another rule having strong confidence is the one associating the Visitor design pattern to the CD smell ($\{V\} \rightarrow \{CD\}$). In this case, we are facing the case where a wrongly implemented DP leads to the introduction of an AS. Both rules have scarce support (i.e., not frequent in the dataset), meaning that such collocations, even if they can be detected and have a strong confidence, are rare.

We also found a rule associating HL to CD (at class level), in both the empirical studies: the confidence in this case was medium-high, about 0.6 and 0.7, and in both cases showed also a strong support. However, both Pearson test and PCA did not find a correlation between the two, meaning that the collocation possibly does not indicate any concrete relationship, but it is spurious.

Instead, PCA identified two groups of AS, that we call clusters, namely (UD, CD) and (FC, HL, GC). The former cluster was confirmed by many other tests: UD and CD have strong positive correlation (Spearman) and strong support, with medium-high confidence. Concerning the latter cluster, we proposed an interpretation of it. The collocation of such smells indicate the presence in the system of a very large, complex component, addressing many responsibilities and being central to the system. The three smells indeed have similar definitions, and violates similar design principles. Knowing that they are strongly correlated is not surprising, however it could be used to discriminate the criticality of the smells: AS who are collocated on the

same component indicate a serious problem that should have top priority with respect to cases where a single smell affects a component.

Future directions concerning this subject can be found at the end of the thesis in Section 8.2.

5 ARCHITECTURAL DEBT EVALUATION

As already outlined architectural smells are a symptom of *Architectural Technical Debt (ATD)*.

ATD is a metaphor used to describe sub-optimal architectural and design choices that seem to be beneficial in the short-term, but reveal to deteriorate the quality of software in the long-term [125]. ATD is a type of the borader concept *Technical Debt (TD)*, which refers generally (i.e., not only addressing the architecture) to the shortcomings developers experience due to the presence of sub-optimal code in the system. The reason behind the term "debt" is that it is compared to the dynamics of financial debt. Indeed, one way to describe technical debt is to divide it in three aspects, measurable independently: principal, interest probability, and interest amount [44]. Principal is the estimated effort to remove the debt-related inefficiencies in the current design or implementation of a software system, while interest is the additional development effort required to modify the software (adding new features or fixing bugs), due to the presence of such inefficiencies. The total technical debt is the sum of the principal, and the product of interest probability and interest amount [44].

Researching and managing ATD is a complex task, mainly because the definition of debt is wide and open to interpretation. However, Verdecchia et al.[246] recently produced a grounded theory about ATD, i.e., they organized the concepts discussed by different authors from the literature about ATD into categories and put such categories in relation to one another. Thanks to their analysis, we can synthesise the concepts related to ATD in the following way.

> ATD instances residing in a system are called *ATD items*, and the main objective of ATD identification is to detect such items. ATD items are generated by a *cause*, which is very difficult to identify a-posteriori. Indeed, approaches for ATD identification focus on the detection of *ATD symptoms* (e.g., architectural smells), that are the measurable aspects of ATD items *consequences*[246].

Some of the consequences of the presence of architectural technical debt in a software project are the delay in the delivery of new functionalities, the difficulty of reusing architectural components in other projects, the compromised maintainability [72] and also the opening towards the introduction of security vulnerabilities [178][222].

Ideally, a project should have a small amount or even no ATD, however it looks like an impossible goal to reach: managing technical debt and assuring software quality are not the current priorities of

software companies [186], mostly because the culture about what is (architectural) technical debt is not widespread and because they lack tools that assist developers to avoid its introduction and to monitor it. Moreover, even if the application of good design and coding practices can help in limiting the growth of ATD, it does not assure the complete protection from it.

Indeed, the current research about ATD is focusing, among the others, on providing means to quantify and monitor ATD items and symptoms. The majority of the proposed identification approaches requires manual analysis or the developer interaction (see Section 7.3.1), instead, little work has been conducted concerning *indexes* to automatically evaluate it [247]. An ATD index is a model (usually a function) which provides a value indicating the amount of ATD in a given software project. Usually indexes take into account code/design level issues and static metrics, and output a number expressed in terms of time or money.

Currently there are a couple of indexes for ATD evaluation, however there are several indexes for TD identification (see Section 7.3.2), i.e., indexes which try to capture different types of TD and do not focus only on ATD. Indexes are useful not only for TD/ATD estimation, but also to compare the software quality of different projects among them. In brief, the value of an ATD index can be exploited by developers to track along time the evolution of ATD in a given project and to compare the project itself with other projects (for instance of the same application portfolio).

In a past work [210], we proposed an index based on AS: Architectural Debt Index (ADI). In particular, ADI takes into account (i) the *Number* of AS detected in a project, (ii) the *Severity* of an AS, where for Severity we mean the cost-solving (see Section 2.2.2) of each instance of AS (an instance of a type of smell, such as CD, can be more difficult to remove with respect to another instance of CD smell) and (iii) the *Dependency metrics* of Martin [150] used for the AS detection. The final *ADI(P)* value quantifies the amount of architectural technical debt present in a project *P*. To the best of our knowledge, the only other index dedicated to the evaluation of ATD is ATDx, defined by Verdecchia et al. [247], which is a model to build ATD indexes based on a set of ATD dimensions that can be collected from source-code analysis.

In this chapter, first we describe our ADI in detail, then we discuss some studies we conducted to evaluate ATD in different contexts and finally we introduce a recent study we made which examine the relationship between AS criticality and cost solving, the two properties used in the computation of the ADI.

5.1 THE ARCHITECTURAL DEBT INDEX

The Architectural Debt Index (ADI) computed by Arcan <of a project P is defined as follows:

$$ADI(P) = \sum_{k=1}^{n} \left(\frac{1}{W}(ASIS(AS_k) * w(AS_k)) * History(AS_k) \right) \quad (10)$$

where:

- n: number of AS instances in a project P;
- AS_k: k-instance of an architectural smell;
- W: the total number of dependencies involved in at least one AS for all the AS in the project;
- $ASIS(AS_k)$: the *Architectural Smell Impact Score*;
- $w(AS_k)$: the *Architectural Smell Weight*, i.e., the number of dependencies associated to the AS_k;
- $History(AS_k)$: the score associated to the trend evolution of the AS_k.

The *Architectural Smell Impact Score*, ASIS, is based on both the estimation of the cost-solving of an AS and the importance of the subsystem where the AS is found. It is defined as the product of the *SeverityScore* (Formula 11) associated to the AS_k smell and the *PageRank* (Formula 12) value of the AS_k, which estimates the importance of the project subsystem affected by the AS_k smell.

With the term *Severity* we generally indicate a metric used to evaluate the cost-solving of an AS. Section 2.2.2 reports all the severities of the AS detected by Arcan, including the ones of the smells considered in the ASIS computation. *SeverityScore* is the quantile of a given Severity.

We use quantiles because the *SeverityScore* and *PageRank* both assume values in the range $[0, \infty)$ and we needed to normalise them in order to exploit them in the formula. We mapped the original values to integer values in the range $[0, 1]$ (low to high respectively), through the $quantile(x)$ function which is the quantile associated to x in a reference dataset consisting of 109 projects of the Qualitas Corpus [238].

The ASIS is defined as follows:

$$ASIS(AS_k) = SeverityScore(AS_k) * PageRank(AS_k) \quad (11)$$

.

Since both $SeverityScore()$ and $PageRank()$ return values between 0 and 1, ASIS represents a $SeverityScore$ weighted by the "importance" ($PageRank$) of the subsystem where the AS appears.

The $PageRank(AS_k)$ estimates whether the AS is located in an important part of the project, where the importance is defined by the value of the PageRank algorithm executed on the dependency graph of the project (to evaluate if many parts (subsystems) depend on the part where the AS is involved). The PageRank, PR is modeled starting from the one implemented by Brin and Page [41], as explained below:

$$PR(v) = \frac{1-d}{N} + d\left(\sum_{k=1}^{n} \frac{PR(p_k)}{C(p_k)}\right) \tag{12}$$

where:

- the vertex v is a node of the dependency graph associated to a project;
- $PR(v)$ is the value of PageRank of the vertex v;
- N is the total number of AS in the project;
- P_k is a vertex with at least a link directed to v;
- n is the number of the p_k vertexes;
- $C(p_k)$ is the number of links of vertex p_k;
- d (damping factor) is a custom factor fixed at 0.85, a default value defined by Brin and Page [41]. It can be changed according to the PageRank value needed for every vertex and its minimum associated level of PageRank.

The PR value is computed only on vertices of the dependency graph of both class and package types. PageRank value PR is used for all the types of AS detected by Arcan, but the PageRank of an AS which involves multiple classes and packages is considered differently, e.g, a CD smell that involves two or more classes or packages. To compute the PageRank when an AS involves more than one vertex, it is necessary to aggregate the data; a method to aggregate multiple values could be to take the maximum PR value of the group.

The PR of all the AS and the max of PR of AS involving multiple classes or packages is computed as follows:

$$PageRank(AS_k) = \begin{cases} \text{If } AS_k \text{ is an AS among classes or packages:} \\ quantile(max_{j=1}^{n} PR(v_j)) \\ \text{If } AS_k \text{ is an AS of a class or a package:} \\ quantile(PR(v)) \end{cases}$$

where v is the vertex (class or package) affected by AS_k, n is the number of classes v_j involved in an AS (among classes) or the number of packages v_j involved in an AS (among packages).

We also report the *q(ADI(P))*, that is its quantification as a score value in a range from 1 to 5, where 5 is the worst rating a project can have (i.e., The higher the ADI value, the higher the debt). Also this value is computed in relation to the reference dataset. If a project is not affected by AS, then the ADI is not computed.

Concerning other approaches for ATD index computation, as already mentioned, Verdecchia et al. [247] developed a method to create ATD indexes named ATDx, which they implemented in a Python tool[1]. ATDx is a data-driven approach that, by leveraging the analysis of a software portfolio, severity calculation of pre-computed architectural violations via clustering, and severity aggregation into different ATD dimensions, provides an overview of the ATD present in a software-intensive system. With respect to ADI, which is focused on AS, ATDx takes into consideration additional facets concerning the software architecture, such as SonarQube [225] rules addressing architectural issues. In this thesis, we do not confront the comparison of ADI with ATDx because it was not an aim of the PhD work. In the future, it could be interesting to understand their differences and compare their outcome when computed on the same set of projects.

5.2 ARCHITECTURAL DEBT INDEX EVALUATION

The following sections report the results we obtained in some studies we conducted about ATD in different kinds of systems and application domains. Our aim was to get an insight about the amount of ATD present in Open Source projects and analyse whether there is some kind of relationship between the presence of ATD and other factors, such as the adoption of third-party software or the application domain of the project.

Indeed, in the first study, we run Arcan on a set of projects before and after their integration with reusable third-party components. Then we show how ATD affects IoT platforms and multi-agent systems along their development history. Finally, we introduce a tool, named Sen4Smells, which is the outcome of an international collaboration we had and which decomposes our ADI looking for the parts of code which contributes the most to the growth of ATD.

Notice that, in some of our studies, we analyse the evolution of the project, i.e., we collect versions (commits, in the case of Git repositories) from a public repository and run Arcan on them. However, we do not analyse each available commit, on the contrary we sample a selection of commits (usually one commit every 30 days). This because architectural changes tend to happen in larger time spans with respect to code changes. A threat to such approach could be that by managing sampling by taking in consideration only the time gap, we would miss out "relevant commits", where the architectural

1 https://github.com/S2-group/ATDx

change actually happens. However, it is not important if we miss relevant commits, because we take into consideration the whole evolution and an architectural change happens during a span (not in a single commit). Similar custom samplings were used in similar context by previous studies [123, 171].

5.2.1 *Impact of Opportunistic Reuse Practices to Technical Debt*

One of the factors that may influence the estimation of technical debt is the reuse of third-party components and their integration into an existing system. As reuse practices may have strong implications in the architecture (e.g. when a platform or a protocol is replaced), we need to estimate the impact on architectural debt derived from the reuse of software components and when a new functionality is added. We focus on a particular type of software reuse, named *opportunistic reuse*, where developers search for Open Source (OS) reusable components in key repositories, such as GitHub, Bitbucket, or Gitlab. In many cases, selecting the right OS components is not easy as many factors, such as licensing, popularity, low code quality (affecting technical debt) may complicate the selection process [228]. In addition, upgradeability plays an important role in the selection of a particular component versus other options. Finally, reusing and integrating third-party components may affect important quality attributes such as safety, integrability, and reliability among others.

A recent study [269] proposed a reusability index based on several metrics to quantify reuse of reusable assets, but the work does not describe a connection to technical debt when reusing components. As to the best of our knowledge there are no works exploring the impact and the connection opportunistic reuse plays in TD. In the research work presented in this section[2] we analyze how this opportunistic reuse trend affects the technical debt ratios in open-source projects and we analyze such TD ratios before and after reusing third-party components, and what are the implications for the software architecture. More specifically, we used SonarQube (see Section 2.5) and Arcan tools to analyze the code and architectural debt ratios in three different applications to investigate the effects of reusing functionality in open source repositories.

CASE STUDY DESIGN We designed and conducted a case study [211] to uncover the impact of opportunistic reuse practices in the quality of systems and analyze the technical debt ratios before and after the integration of open-source software (i.e. understood as new functionality) in three existing systems. Therefore, we followed the approach of *exploratory case studies* [264] as a way to explore and identify the

2 A publication was extracted from this study [50], done in collaboration with Rafael Capilla, Valentina Lenarduzzi and Tommi Mikkonnen.

overall picture of opportunistic reuse practices. To this aim we raised the following research questions:

- *RQ1: Which are the most difficult aspects to integrate new functionality found opportunistically in open source projects?* We study the effort and the changes required by a developer when has to integrate third-party components, so we can compare the integration effort to develop components from scratch, if necessary.
- *RQ2: How code debt is affected after the integration of reusable assets found opportunistically?* We wanted to investigate to what extent opportunistic reuse practices have a significant impact on TD ratios and the number of new smells in the code.
- *RQ3: How architectural debt is affected by reusing open-source software?* Like in the previous research question, we are interested to analyze how architectural debt (ATD) is affected when new components are introduced in existing architectures and what changes induce.

Following, we describe the context of the exploratory study including the three case studies we used and the repositories we chose to reuse the components.

CASE STUDY 1: BIKEAPP This case study belongs to a bicycle hiring system (BikeApp) developed at the Telecommunications School from the Technical University of Madrid (UPM) between February-May in 2020. The system promotes sustainable transportation in Mad rid renting bikes around the city and using a mobile app. The system is built around an API that uses a Client/Server style and a persistence layer (Java Persistence API - JPA) to access the database. The server side uses J2EE and servlets (i.e. Servlet) supported by Apache Tomcat 9.0. The client-side uses a multi-window approach based on Java Swing. The source code of this small project is only 2.500 SLOC, as we did not include the third-party libraries required to develop the software. We used the Visual Paradigm tool 16.1[3] to reverse the architecture of the system, shown in the class diagram in Figure 5.1, which encompasses 52 classes. In the Figure, we can see the main BikeApp class (in brown color), while the UML components (shown in grey color) provide support for the `hsqldb,jackess` libraries for accessing the SQL database and a connection to the `EclipseLink` component aimed to link database objects to Java objects handled by the JPA persistence layer. Also, the client uses the `jxmapviewer` graphical library which receives the location of the bikes encoded in JSON format.

3 https://www.visual-paradigm.com/

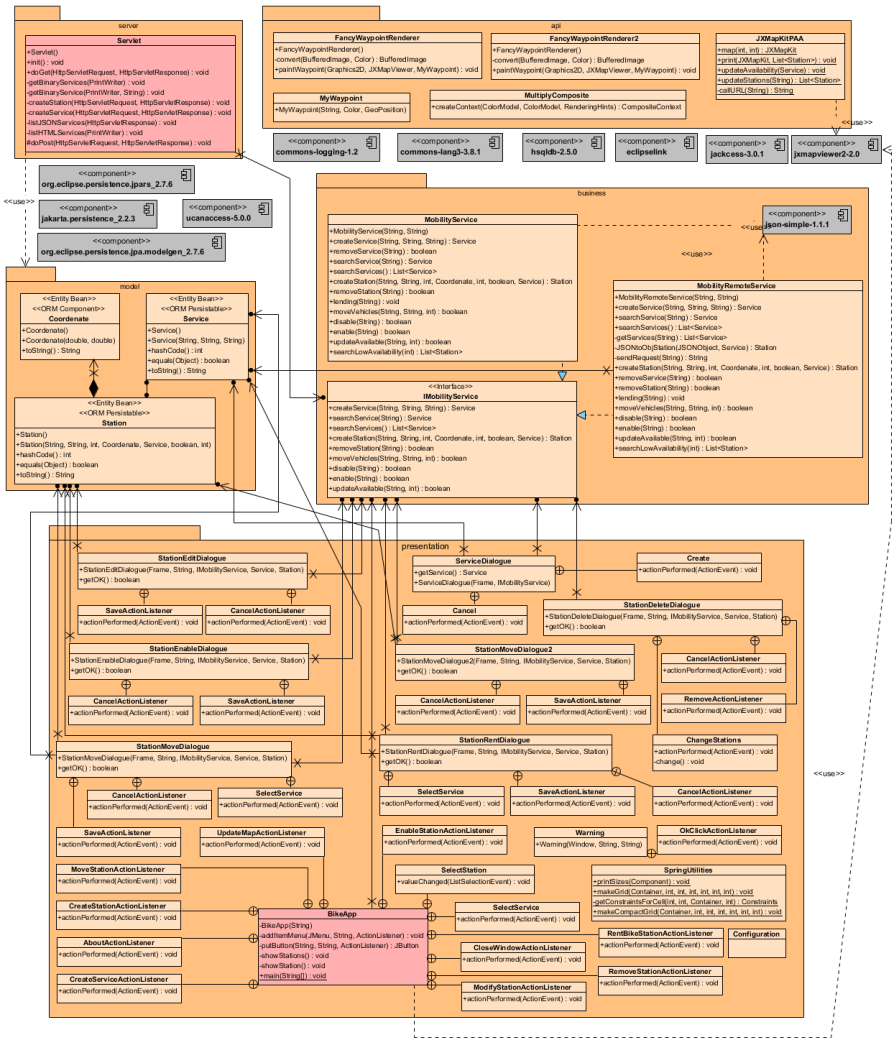


Figure 5.1: Reversed architecture of the BikeApp software

CASE STUDY 2: TEAMMATES The second case study is an OS system called TEAMMATES [4]. We used version 7.6.0 for our analysis. TEAMMATES is a free online tool for managing confidential peer evaluations for student team projects. The students can evaluate their performance anonymously in team projects and search/view reports of their feedback and evaluations. TEAMMATES was designed to provide powerful peer feedback and peer evaluation mechanism with a very high degree of flexibility. TEAMMATES runs on the top of Google App Engine, using cutting-edge cloud technologies and benefits from the infrastructure that power Google's applications.

CASE STUDY 3: KURKI This case study addresses an IT system Kurki[5], used for managing students participating in courses, their course accomplishment, and the necessary operations to provide grades and other related information. Kurki is a web application, where then frontend relies on HTML, CSS, and JavaScript, and the backend includes Java code for implementing application-specific functions, a web server for processing requests and responses, and an SQL database for storing the information. Designed at the Department of Computer Science, University of Helsinki, Finland, the system has been deployed to use in the 1990s, and many of the design choices still present in the system reflect the state-of-the-art of those days. Over the years, portions of code have disappeared when people working on the system have left the project, resulting in updates in certain parts of the system. Furthermore, numerous developers have participated in the project. Internally, the system includes some legacy code, but things that are related to operating it has been upgraded to today's standards. For instance, to deploy the system, Docker is used, and doing a git push to the master branch and running an associated script are enough to deploy the system.

REPOSITORIES To search for reusable components, we selected the following four OS repositories: *Maven, GitHub, SourceForge, and GitLab*. We based our selection on the following criteria: (i) explanation and documentation of the component including a Javadoc file, (ii) compatible license and version with the target software, (iii) existence of an import file to facilitate the integration process, (iv) access to the source code, (v) functionality required (vi) popularity and (vii) dependencies to other libraries. Some of the aforementioned items are also discussed in [228].

DATA COLLECTION AND ANALYSIS In this step of the study, we performed the following tasks: (i) we run SonarQube on the three projects and we collected the technical debt ratios, the number of

4 https://teammatesv4.appspot.com/web/front/home
5 https://github.com/UniversityOfHelsinkiCS/kurki

code smells and issues, (ii) we run the Arcan tool to compute the ADI (see Section 5.1) and detect three architectural smells, namely Cyclic Dependency (CD), Hub-Like Dependency (HL) and Unstable Dependency (UD) (see Section 2.2), for all the analyzed projects, (iii) we sought in the four repositories for new functionalities to extend the three projects, and (iv) once the components found were integrated into the three projects, we run again SonarQube and Arcan to measure the technical debt and the other quality ratios provided by the tools. The information about the number of components found and reused is described in Tables 5.2 and 5.3. The results of the technical debt rations after reuse are shown in Table 5.5 shows. The search and integration efforts were computed manually by two researchers.

RESULTS First, we describe the outcome from SonarQube and Arcan tools for the three case studies before extending the functionality with third-party components, such as we describe in Table 5.1. The results we computed include the technical debt ratios (TD ratio), code smells (Smells) and issues (Issues) provided by SonarQube, the number of Architectural Smells (AS) and the Architectural Debt Index (ADI) provided by Arcan and the quantification of ADI (q(ADI)) as a score value in the range from 1 to 5. In order to select and search for reusable assets, (1) we simulated a scenario where the project's team wants to extend the functionalities of that project basing on potential client requirements; (2) we made a list of the new desiderata, and (3) we searched for them in the repositories. The desiderata was brainstormed by three researchers of our team.

Table 5.1: Results from SonarQube and Arcan before reuse

	SonarQube			*Arcan*		
Case studies	**TD ratio**	**Smells**	**Issues**	**AS**	**ADI**	**q(ADI)**
BikeApp	5.2	248	253	0	-	-
TEAMMATES	0.6	2087	2905	7	8.0	5
Kurki	1.0	504	536	3	5.0	5

Table 5.2 summarizes the number of components we found for each project and the different repositories. For the **BikeApp** case study, we searched for the following functionality: (i) encode the data between the server and the clients, (ii) provide new functionality to geolocate the bikes, and (iii) a calendar. For the **TEAMMATES** project, we looked for the following functionality: (i) a voting system, and (ii) an event manager. Finally, in the case of **Kurki** as it is similar to TEAMMATES, we decided to include only the event manager functionality so we can compare differences in the TD ratios. For the three projects, we used different keywords in Google as the search string to search for reusable assets in the four repositories.

Table 5.2: Number of reusable components found in open-source repositories

Case studies		Maven	GitHub	SourceForge	GitLab
BikeApp	Encode data	115	170	9	13
	Display geolocation	7867	92	5	0
	Calendar	1648	111	8	2
TEAMMATES	Voting system	17	14	92	9
	Event manager	561	80	25	78
Kurki	Event manager	561	80	25	78

In Table 5.3 we show the components we found and reused for each project and from which repository we selected those components. The criteria to select a component from a particular repository was based on (i) the description of the functionality of the component, (ii) compatibility of licenses between the software and the reused component (iii) access to the source code, (iv) compatibility of the version of the reused asset with the existing project and dependencies to third-party libraries, (v) facility to import and configure the component, and (vi) existence of a Javadoc file explaining how to use the components found. Other criteria such as popularity or project releases can be also considered. The developer can select one or several of these criteria to find the most suitable asset.

Table 5.3: Reusable components selected in open-source repositories

Case studies	Maven	GitHub	SourceForge	GitLab
BikeApp	Base64	—	Base64	—
	GoogleMaps	GoogleMaps	—	—
	JDatePicker	JDatePicker	—	—
TEAMMATES	Voting-Reward	Voting-Reward, Voting-System	—	—
	—	FullCalendar	—	—
Kurki	—	FullCalendar	—	—

SEARCH AND INTEGRATION EFFORT In the following, we explain how we integrated the components found in the repositories and the effort taken. We assume we consider the time needed starting from the initial search for each component and project until a component was found and reused, and the time required to integrate the component into the project and check that the application does not contain compilation errors and the new functionality is ready to be used. Regarding the search effort, we run Google queries for each new functionality and we used the initial set of results to refine the search in each of the four repositories. If the selected component does not serve, we refine the query in each repository or we look for the next pop-

ular component. Each repository has different facilities to perform a refined search (e.g.: keywords, categories, or search string).

BikeApp: In this case study we reused three components. For the **Base64** component and based on the criteria we defined, we used the search string *"send java data encoded safety"* in Google and we got around 7 million responses, but the two first pages gave us 21 responses including Base64. We looked for comments about pros and cons and we analyzed the functionality of the component as well as the documentation and other comments. Then, we looked for the selected component in the four repositories and eventually selected the one from SourceForge because it does not require dependencies to other libraries. All this effort took around 4.5 hours. The integration was done by importing the asset directly without using Maven. We downloaded those libraries required by the client and the server and we modified the corresponding Classpath file. Then, we modified the Servlet class (on the server-side) to gather the HTTP request needed by the Base64 component and we added two new methods to integrate the reused component. On the client-side, we modified the HTTP request to process the data received. We also created a new class to configure the app and select the type of encoding (i.e. HTML, JSON, Base64). We spent between 1.5 and 1.7 hours in the integration effort.

Regarding the search of the **GMaps** component we looked for an asset that provides the location of the bikes, so the search string we used was *"get gps coordinates in Java"*. As a result, we got 1.8 million references. On the two first pages, we found 4 references, and the asset chosen was found in Maven and GitHub repositories. Based on the same criteria as was used in the first case, we selected the component from the Maven repository because it provides compatibility information and which dependencies to other libraries are needed. GitHub does not provide this information in such a clear way. The effort spent in searching the right component, and the integration effort are shown in Table 5.4.

The integration of this asset required the modification of the class that manages the events to select a hiring bike point and provide the right address encoded in JSON format. We also needed to register into the Google Cloud platform in order to access the Geocode API used to decode the addresses of a GPS location. In addition, we had some connectivity problems during the testing of the reused asset that we solved by installing an additional component required by Google.

Finally, we did the same for a calendar we needed to integrate using Java Swing and the search string we used was *"Java swing calendar"*. We got around 5 million results. From the first two Google pages, we found a reference to the **JDatePicker** component, which was available in Maven and GitHub repositories. We selected the component

from GitHub because it does not exhibit dependencies to other components and due to the availability of a tutorial and examples of use. We imported the component and we only needed to modify the class that manages the event that activates the calendar. Similar to the other components, the effort spent in the search process and integration as well are shown in Table 5.4.

TEAMMATES: We did the same for the TEAMMATES projects reusing 2 components. Regarding the **Voting system** component we used the search string *"Java student voting systems"* and we got around 9.6 million answers. Screening the first two Google pages, we found two similar components (i.e. voting reward and voting system) in different repositories and reused the *voting system* component according to our criteria but the selection was mainly based on because is an independent Java project that does not require dependencies to external projects and also does not require changes to be integrated with other software. The second component is an extended calendar including an event manager for appointments. We used the search string *"event manager calendar in Java"* and we got 9.8 million results but we reused the **Event manager** component from GitHub because of the examples provided by the third-party developer, documentation of use, as well as references from the other repositories to this component. The results about the components found and reused are shown in Table 5.3, while the effort reusing and integrating both components is available in Table 5.4.

Kurki: As this is a similar project like TEAMMATES, we decided to integrate the same **FullCalendar** component previously reused so we can compare the trend of the TD ratios of both projects and observe if there are significant architectural differences. In this case the search effort is equal to 0 and the integration effort compared to TEAMMATES just a little bit higher due to differences in the technologies used in both projects. Table 5.4 describes the summary of the search and integration efforts for the different components.

Table 5.4: Search and integration efforts of the reused assets (hours)

Case studies	Assets	Search effort	Integration effort
BikeApp	Base64	4.5	1.7
	GoogleMaps	5	5.5
	JDatePicker	6	1.1
TEAMMATES	VotingSystem	6.5	1.4
	FullCalendar	6	2
Kurki	FullCalendar	0	2.2

Architecture after reuse: Finally, we reversed the new architecture of BikeApp including the new components. As we can see in Figure 5.2, the new classes and components are shown in green color while the entities required by these elements are displayed in light yellow.

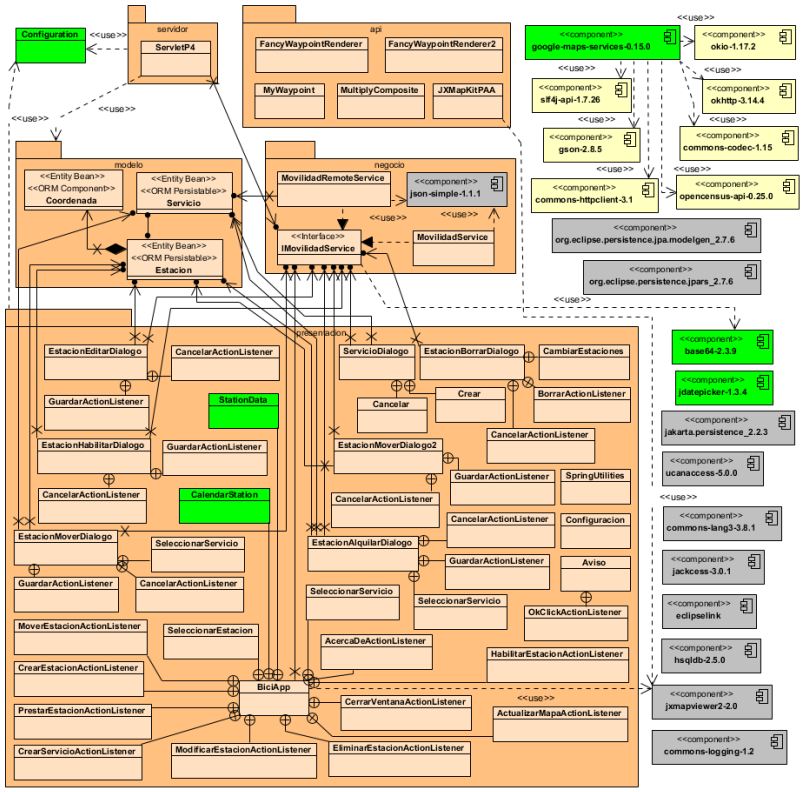


Figure 5.2: Final version of the reversed architecture of the BikeApp software

Some of the new entities in green were created by us to invoke the three new components. We added 13 new classes and components for the new functionality reused, that is an increment of 17% of elements from the original design. It might be possible that reusing different components could have different numbers in terms of new functionality, but what is more important is that the architectural style did not change after reuse.

DEBT RATIOS AFTER REUSE In Table 5.5 we describe the results of the technical debt ratios from SonarQube and the ADI index from Arcan after reuse. As we can observe we computed the TD ratios for each project taking into account the SLOC of the new components in a cumulative way (i.e. the TD ratio of the last component of each project includes the previous ones). In the case of the **BikeApp** project, the trend of the TD ratio observed decreased from 4.5 to 1.7. However, the number of smells and issues for SonarQube increased from 248 and 253 (see Table 5.1) to 3551 and 3759 respectively.

From the results of Arcan, we did not find any architectural debt in the original version of the project, but this debt increased after reuse. Our results show many architectural smells (30 for Base64, 64 for BikeApp-GoogleMaps, and 66 for BikeApp-JDatePicker) with a high q(ADI) value. In particular, Arcan identified the following ar-

chitectural smells: 1 UD, 43 CD, 3 HL for **Base64**; 18 UD, 43 CD, 3 HL for **GoogleMaps**; 19 UD, 44 CD and 3 HL smells for **JDatePicker**. Concerning the specific types of smell detected, Arcan only identified smells of type Cyclic Dependency. Hence, the values reported in Table 5.5 refer only to that type of smell.

Regarding the **TEAMMATES** project, we observed that due to the size of the project (i.e. 128k SLOC) and the small size of the reused components, the TD ratios shown in Table 5.5 are the same. Only the number of smells and issues varied. In the case of the voting system, the number of smells grew from 2087 to 2136 and the number of issues increased too from 2905 to 2977. For the event manager component, we got the same number of smells as for the voting system, the increment is insignificant, that is 2979 issues. Regarding Arcan results, the ADI value (both ADI and q(ADI)) did not change since the original project versions. In particular, the ADI value, which equals 12.0 for both extensions, is very close to the ones of BikeApp-GoogleMaps and the BikeApp-JDatePicker. The only change happened in terms of detected smell types: respect the original version, which counted 5 UD and 2 CD, the extended ones show one more UD (for a total of 4) and one less CD (for a total of 3). About **Kurki**, we observed the typical increment in the number of code smells and TD issues compared to the initial project but the TD ratio decreased from 1.0 to 0.6. This is caused because Kurki is not so big a project like TEAMMATES and doubling the number of SLOC after including the new functionality led to a significant reduction of the TD ratio. Regarding Arcan results, as happened for TEAMMATES, the ADI value (both ADI and q(ADI)) did not change from the original project version. Also, the number of smells by type remained the same, i.e., one smell per type.

Table 5.5: Technical debt and architectural debt ratios after reuse

		SonarQube			*Arcan*		
Case studies	**Assets**	**TD ratio**	**Smells**	**Issues**	**AS**	**ADI**	**q(ADI)**
BikeApp	Base64	4.5	365	383	30	1.0	3
	GoogleMaps	1.7	3494	3701	64	11.0	5
	JDatePicker	1.7	3551	3759	66	10.0	5
TEAMMATES	VotingSystem	0.6	2136	2977	7	12.0	5
	FullCalendar	0.6	2136	2979	7	12.0	5
Kurki	FullCalendar	0.6	735	678	3	5.0	5

FINDINGS In this section, we describe our findings answering the three research questions.

We can say that the most complex issues we found during the integration of the reused components are the following. First, to identify

which elements of the target project which *1)* needed to be modified *2)* are influenced by the architectural style of the project and *3)* are underlying technologies used to implement the architectural style of the project. For instance, for BikeApp and TEAMMATES projects using the model-view-controller (MVC) style we needed to identify the right Web technologies (e.g. BikeApp uses Swing to implement the "view" while TEAMMATES uses Angular).

Second, seeking the right component according to the architectural style and technologies used in the target project, as in other cases the integration of a reused component using different technologies may lead to performing another search. This happened when we integrate the **voting system** component for TEAMMATES. Also, in the case of TEAMMATES and Kurki, we observed certain differences in the integration effort caused by the different technologies used in both projects.

Third, other minor issues like the invocation of the new component or the creation of an instance of the object can be solved by common programming techniques. Independently of some integration problems, we followed the criterion of ease of integration based on a small number of dependencies to third-party libraries and the compatibility of the reused component with the existing application. In brief,

> ***RQ1: Which are the most difficult aspects to integrate new functionality found opportunistically in open source projects?*** The most complex issues we found during the integration are the identification of the parts of code to be modified in the target project, the identification of the reusable component itself and the technological gap that might arise between the target project and the reusable component.

From the analysis of the code debt variation before and after the reuse, we observed from Table 5.5 that the Sonarqube Technical Debt ratio decreases if we reuse large-scale components like the **Google Maps** one. On the contrary, if we start from a large software project and the sizes of the components are small, the code debt ratio provided by SonarQube remains almost the same. In addition, in the majority of the cases, the number of SonarQuve smells and issues increase, but we found one case (i.e. the voting system component) where the number of issues decreased a bit. Another aspect not covered in this study but worthy to be investigated is the criticality of the new smells and issues and not counting only the number of instances.

> ***RQ2: How code debt is affected after the integration of reusable assets found opportunistically?*** The TD ratio appears to decrease when reusing large-scale components and to stay the same with

small components. However, the number of SonarQube smells and issues increases.

Finally, concerning the impact of reuse on architectural debt, our results show that reuse has an impact in terms of architectural smells and consequently of architectural debt. In general, from what we observed in Table 5.5, architectural debt increases after reuse, and in particular when reusing large-scale components (**Google Maps** and **JDatePicker**). In the case of project TEAMMATES, where the number of AS does not change after reuse, the debt increases too, meaning that the AS worsen in terms of *severity* (see Section 2.2.2), e.g., they grow in size and affect more components (classes and packages). The only project which was not affected by reuse is Kurki, whose number of architecture smells (AS) and ADI values remained the same. Instead, in some cases, we needed additional search effort to find suitable components aligned by the technologies supporting the architecture of the project.

RQ3: How architectural debt is affected by reusing open-source software? The number of architectural smells and the architectural debt tend to increase after reuse.

THREATS TO VALIDITY Some factors might have influenced the results reported in our study. We discuss the main threats to validity and how we mitigated them according to Yin's guidelines [264]. To enable the study replicability, all exploited material can be found in the replication package[6]

Construct Validity. We adopted the default set of collected measures considered by the SonarQube model since practitioners are reluctant to customize the built-in quality gate and mostly rely on the standard set of rules [245]. Also, we have tried as well as possible to replicate the conditions adopted by practitioners that use this tool, although we are aware that the detection accuracy of some rules may not be precise.

Internal Validity. SonarQube detected duplication of the same issue, reporting the issue violated in the same class and in the same position but with different resolution times. We are aware of this fact, but we did not remove such issues from the analysis since we wanted to report the results without modifying the output provided by SonarQube and introducing other biases in the study.

External Validity. We analyzed three case systems trying to select different projects with different characteristics. However, we are aware that other projects might present slightly different results. We have considered for architectural debt detection only the AS detected by the Arcan tool. We could have different results by considering other

6 https://github.com/CCS-repository-public/techdebt-2021

AS, but these smells based on dependency issues are certainly particularly critical for a project. In the future, we plan to extend the works by considering additional types of AS.

Conclusion Validity. We can rely on the two tools we selected. SonarQube is one of the most popular static analysis tools largely adopted both in academia [139, 140] and in industry [245]. Validation of Arcan results is discussed in Chapter 3.

FINAL REMARKS To the best of our knowledge, this is the first work that examines the impact of TD when reusing software components opportunistically in different software repositories. Our main findings show that for larger projects the TD ratio provided by SonarQube remains stable or decreases, but in most cases, the number of code smells and TD issues increase. The same happens when we add a large component to a small project. We also evaluated the projects' architectural debt and the number of architectural smells before and after reuse. Similar to SonarQube, Arcan detected more architectural smells after reuse, which resulted in increasing architectural debt. One interesting outcome is that in the most affected project (BikeApp), the most common type of smell is Cyclic Dependency, suggesting that developers should particularly pay attention to this specific smell while reusing software. This work, being a case study, does not provide conclusive evidence of the effect of opportunistic reuse on TD. Therefore, further research is needed on the topic, to better understand the right approach to measure TD in relation to opportunistic code reuse.

5.2.2 *Evaluating the Architectural Debt of IoT Projects*

IoT software development is known to be different from the development of other kinds of applications [59]. Development of IoT applications, as outlined by Patel et al. [189], is particularly challenging because it involves dealing with a wide range of issues such as "lack of separation of concerns and lack of high-level of abstractions to address both the large scale and heterogeneity" and other issues. Hence, Software Engineering technologies and tools are crucial to design, develop, deploy, and maintain high-quality IoT systems [129] and it is important to capture the software engineering needs in the IoT context.

While many works have done on evaluating ATD of non-IoT related systems, according to our knowledge this kind of analysis has not been yet explored for IoT systems. We wanted to start exploring this field, hence, in this section[7] we focus our attention on evaluating the ATD of 4 IoT platforms. An IoT platform is a set of software components that support developers in implementing IoT architectures. A platforms allows to build applications, connect to devices and remotely collect data, secure connectivity, and execute sensor management. To evaluate ATD we use the Arcan tool. We report the results obtained that show that also IoT platforms can be subjected to ATD and suggest developers to take care of ATD by conducting periodical refactorings.

STUDY DESIGN We introduce the design of this study and the following Research Questions we aim to answer:

- *RQ1: Which are the most present types of AS in IoT platforms?*
- *RQ2: What can we observe according to architectural debt of IoT platforms?*

To answer the two RQs we evaluate the ATD in terms of the AS and the Architectural Debt Index (ADI) computed through Arcan.

7 A publication was extracted from this study [81]

Table 5.6: Analysed projects - Metrics

Project	#Com.	First commit	Last commit	LOC first	LOC last
crate	432	26/06/2013	11/01/2021	208	470982
thingsboard	238	05/12/2016	12/01/2021	103106	178221
blynk-server	74	28/04/2018	23/12/2020	47382	54692
paho.mqtt.android	43	27/08/2015	04/06/2020	5591	6405

Table 5.7: Analysed projects - Additional information

Project	Domain	Github
crate	Distributed database	https://github.com/crate/crate
thingsboard	Data analytics platform	https://github.com/thingsboard/thingsboard
blynk-server	Home automation platform	https://github.com/blynkkk/blynk-server
paho.mqtt.android	Device connectivity framework	https://github.com/eclipse/paho.mqtt.android

Since we have large experience [21][210][79] in analyzing the ATD in open source projects, but not in IoT systems, through the answer to these RQs we aim to analyze the ATD of IoT systems, in order to provide some preliminary hints to the developers. In case ATD is present or specific AS are identified in the systems, developers have to pay attention to these problems to prevent them or remove them as soon as possible.

ANALYZED PROJECTS We selected four IoT Java projects from the list made available by Corno [59]. In particular, we selected the projects written in Java, the programming language supported by Arcan. All projects are hosted on Github, among the Open-Source and popular ones in the Github community (highly ranked with Github stars). Table 5.6 and 5.7 show the main project characteristics: names, the number of analysed commits, the considered time period (date of the first and last commit), size expressed in Number of Lines of Code (LOC) both for the first and last commit, the application domain and finally the Github url. Concerning the commit analysis, we considered only commits pushed or merged into the master branch, starting from the beginning of the commit history and by sampling one commit every 30 days.

COLLECTED DATA We collected data about CD and HL, both at class and package level, and UD. We also computed for each analysed version its ADI value.

ANALYSIS In order to answer our research questions, we conducted two kinds of analysis on the architectural debt data (number of AS and ADI) collected with Arcan. *First, we extracted a set of statistical metrics* (mean, standard deviation, minimum value, maximum value) to ease the interpretation of the results. All metrics are evaluated with respect to the analysed time period, i.e., the data extracted from the

considered commits. In this way, we can compare the results of the different projects, even if they were calculated on time periods of different length.

We also conducted trend analysis to understand how ADI and AS evolve overtime. We exploited the *Mann-Kendall test* (see Section 4.1), which is a non-parametric test able to assess if there is a monotonic upward or downward trend of the variable of interest over time. In the context of this study, given the number of AS and the ADI value for each commit, the test is able to compare the values across history (i.e., the commits ordered by time of creation) and determine whether, along time, the number of AS and ADI increases/decreases or does not show a trend. If a trend is present, it can be the first clue that the presence of AS and ADI has a relationship with other kinds of variable, i.e., the maturity of the project, the seniority of the developers, the development practices adopted by the developers and so on.

Finally, we conducted a manual validation of the results of the tests. In particular, we collected the commit comments attached to Github and the available release notes. This was useful to offer an interpretation of the results of the single projects and to acquire information useful to compare the different projects among them.

FINDINGS In this section, we report the results of our analysis and also the answers to our research questions. Table 5.11 reports results of the distribution analysis conducted on the 4 projects. The statistics are evaluated on the number of AS, also divided by type (CD, HL, UD), and on ADI, measured for each commit during the considered time period. The project with the highest mean number of AS is Crate (≈ 425), which has also the highest mean value of ADI (≈ 35). On the other hand, Blynk-server has the lowest AS and ADI mean values. Hence we can provide the answer to the first RQ:

> ***RQ1: Which are the most present types of AS in IoT platforms?*** The most present types of AS (on average) are UD (see values of Blynk-server and Paho.mqtt.android) and CD (see values of Crate and Thingsboard). The less present AS is HL.

We also ran the Mann-Kendall tests to analyse the trend of the same variables (CD, HL, UD, AS and ADI). Table 5.12 reports the results only of the significant cases, i.e., with p-value < 0.05. The table also indicates whether the *trend* is increasing ($+$) or decreasing ($-$).

We now put in relation the results of the two analysis and provide a brief discussion of the architectural debt of each project. Each of the following figures depicts the ADI trend (y-axis) of the projects, computed for each commit (x-axis).

Blink-Server: The ADI trend is decreasing, as also pictured in Figure 5.3. Another decreasing trend is the number of CD, which is also the type of smell most present in this project, i.e., which determines

Table 5.8: Distribution analysis results

Metric	Blynk-server	Crate	Paho.mqtt.android	Thingsboard
CD mean	1.73	379.50	11.52	5.00
CD std.dev	1.58	435.95	3.60	0.00
CD min	1	4	6	5
CD max	5	1456	20	5
HL mean	1.00	2.37	1.00	1.00
HL std.dev	0.00	1.20	0.00	0.00
HL min	1	1	1	1
HL max	1	5	1	1
UD mean	3.75	52.03	14.34	2.00
UD std.dev	1.90	34.85	3.19	0.00
UD min	1	1	10	2
UD max	11	143	24	2
AS mean	3.54	425.19	6.93	25.97
AS std.dev	3.26	468.11	1.98	6.37
AS min	1	1	1	17
AS max	17	1568	8	45
ADI mean	0.92	34.84	5.47	2.94
ADI std. Dev	1.03	28.74	2.37	1.30
ADI min	0	0	0	1
ADI max	5	199	8	8

the most the value of ADI. This means that the continuous removal of CD contributes to the progressive reduction of the architectural debt estimation. In general, this project on average has the smallest number of AS, the smallest values of ADI and the best history in terms of architectural quality improvement. We manually checked the changelog available on Github and realised that one thing that distinguishes this project from the others ones is the (good) habit, of the development team, of conducting periodic refactorings. The evidence of this behaviour can be easily traced from the changelog titles ("fixes", "improvements", "cleanup"). One example, the comment to the release 0.38.5 includes the statements "*Tests speedup and refactorings*", and also "*Singletons for Hardware handler*": Singleton is the name of a design pattern [88], an additional signal of the fact that they particularly took care about design quality and best practices. However, this is also the smallest project in terms of LOC and also the project with the shortest history. This could have influenced the final results.

Table 5.9: Mann-Kendall test results

Project	Variable	p-value	Trend
Blynk-server	CD	0.002	-
Blynk-server	ADI	1.87E-07	-
Blynk-server	AS	1.66E-14	-
Crate	CD	0	+
Crate	HL	0	-
Crate	UD	0	-
Crate	AS	0	+
Paho.mqtt.android	AS	0	+
Paho.mqtt.android	ADI	0	+
Thingsboard	ADI	0.004	+

We also checked the changelog of the commits corresponding to the sudden drop of the ADI value from 5 to 0: in this case we have not found references to any refactoring activities, however they report the "*Drop* [of an] *old HTTP API*". Our idea is that the developers changed a relevant part of the code related to a legacy, eroded system's API and this substantially improved the ADI value.

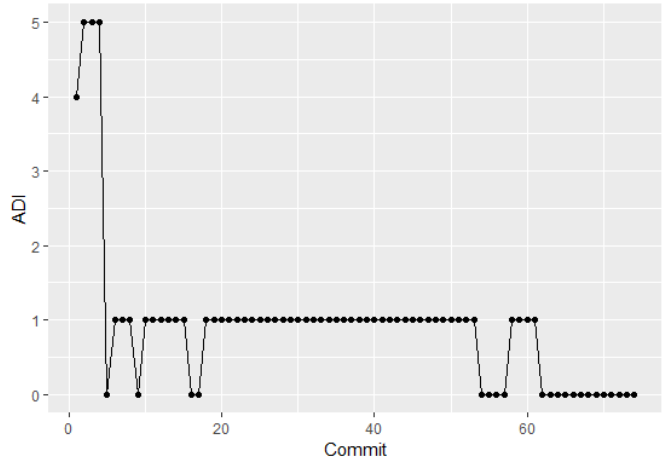


Figure 5.3: Evolution of ADI value - Blynk-server

Crate: This project showed no ADI trend. From Figure 5.4, we can see why: instead of following a trend, the data are distributed in periods where ADI oscillates between fixed values. In particular, in the first $\approx$ 250 commits it ranges from 20 to 50, then it starts increasing and oscillating between 20 and 200, and finally, in the last commits, it is subjected to a strong decrease, ranging from 0 to 20. The oscillatory behaviour might be explained by the fact that the development

team exploits a code quality checker, named Spotbugs, which they could use periodically to clean up the code. However, Spotbugs does not address architectural or design concerns. Hence, we checked in the project changelog, but we did not find indications to interpret the evolution of architectural debt for this project. In brief, this project has the highest mean ADI value, the larger number of AS, but we have no clear information about the architectural debt history.

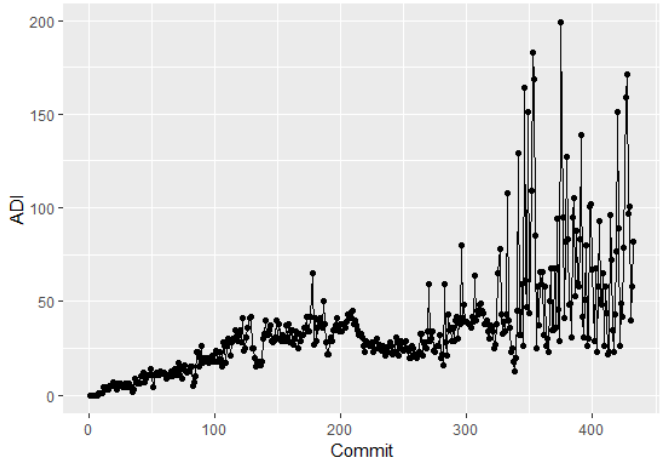


Figure 5.4: Evolution of ADI value - Crate

Paho.mqtt.android: This project's ADI trend is increasing (Figure 5.5). The few amount of commits allows us to follow the evolution of its subsequent plateaus. Even if computed on few commits, its mean ADI is the second highest among the four projects. We manually checked the commits' comments, but we did not find any interesting clue behind why this project tends to increase its ADI.

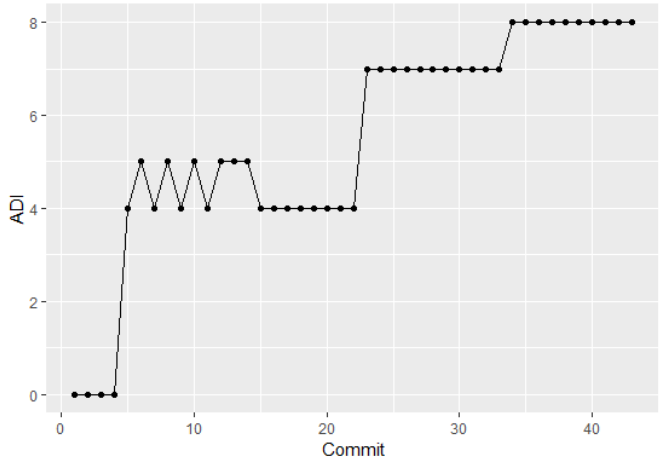


Figure 5.5: Evolution of ADI value - Paho.mqtt.android

Thingsboard: Also this project ADI is increasing during time, even if it cannot be grasped directly from Figure 5.6: the value oscillates a

lot in the space of few commits. We manually checked the commits and release comments, and found out that the development team maintains two distinct versions of the project: thingsboard 3.X.X and thingsboard 2.X.X, and both version releases are periodically merged into master (the git branch that we consider during the analysis). This explains why the ADI oscillates in this way.

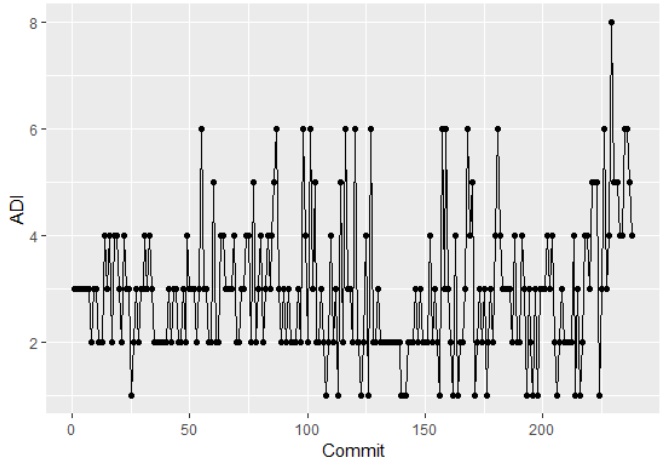


Figure 5.6: Evolution of ADI value - Thingsboard

To conclude:

RQ2: What can we observe according to architectural debt of IoT platforms? the analysed projects show an increasing ADI trend, with the exception of project Blyink-server, whose periodical refactorings allowed the control of the debt. Thus, developers of IoT systems should take care of the appearance of architectural debt and schedule frequent maintenance activities.

THREATS TO VALIDITY This study presents some threats to validity, we describe them according to Yin's guidelines [264].

Concerning the *construct validity*, we chose to use our ADI, and the choice highly influence the results, because the index is based on AS and other kinds of indexes may take into account different facets related to ATD. However, we chose to use ADI because *1)* we wanted to focus exclusively on ATD and not on other kinds of debt, *2)* we already had it implemented in Arcan 3) at the moment of the study the implementation of ATDx [247] was not available.

Concerning *internal validity*, the consulted code commits may have been misleading, providing poor insights about the actually committed activities. However, code comments are considered a (limited) but valid source of information and have been exploited in other studies [262][206].

Concerning *external validity*, we analysed few projects, all written in Java. However, our aim was to conduct a preliminary study and

focus on the manual validation of single commit comments, which would have not been feasible with a large pool of projects. In the future, we aim to extend our analysis to more and different types of projects. Another threat concerns the considered types of AS: the choice of other smells may lead to different results. In the future, we want to extend ADI with more types of AS and subsequently extend the analysis on such smells.

Concerning *reliability*, we amply discussed the validation of Arcan results in Section 3.

FINAL REMARKS We evaluated the architectural technical debt of 4 IoT platforms, during their development history, for a total of 787 analysed commits. In particular we computed the Architectural Debt Index (ADI), based on the detection of 3 different AS. All the considered projects are affected by architectural debt, in particular project Crate. However, we found evidence in the changelog of project Blynk-Server that periodical refactorings are effective in keeping the debt under control: indeed, it is the only project with a decreasing ADI trend along its development history. Another finding regards the type of architectural debt discovered in the projects: from our analysis we found out that Cyclic Dependency and Unstable Dependency are the most present types of AS. However, this is a preliminary study conducted on few projects, and we analysed only Java code. Moreover, we did not keep in consideration the fact that the analysed projects/platforms came in different types and had different purposes, and whether the impact of AS changed depending on such types.

5.2.3 *Evaluating the architectural debt of agent based systems*

As in the case of IoT projects, to the best of our knowledge, there are no studies about ATD in MultiAgent Systems (MAS) and MAS development platforms. Anyway, the MAS community is deserving to performance [169], scalability [8], security [69] and in general Software Engineering aspects more and more attention in the last years. Trend which is confirmed for example by the creation of the dedicated Software Engineering area of interest at AAMAS (International Conference on Autonomous Agents and Multiagent SystemsInternational Conference on Autonomous Agents and Multiagent Systems), workshops focusing on Software Engineering topics (for example EMAS (Engineering Multi-Agent Systems) and AREA (Agents and Robots for reliable Engineered Autonomy) [51]), and works on Engineering MultiAgent Systems (for example [157, 266]). It looks like it is only a matter of time before other Software Engineering topics will be faced by the MultiAgent Systems community too.

In this study[8] , we aim to analyse four well-known and largely adopted MAS development platforms (Jade, Jason, Jadex and Netlogo) in order to evaluate their architectural debt: since these platforms are used by many developers and have been released in many versions in a quite long lifespan, we are interested in evaluating if they suffer of ATD, so that in case to provide to their developers useful hints to improve their quality.

Also in this case, We exploit Arcan to compute the Architectural Debt Index (ADI).

Our results show that the considered systems suffer from ATD, thus their developers should be aware of it so that to be able to manage these issues in future releases.

STUDY DESIGN We introduce the design of this study and the following Research Questions we aim to answer:

- *RQ1: Which is the most present type of AS in MultiAgents Systems platforms?*
- *RQ2: What can we observe according to architectural debt of MultiAgents Systems platforms?*

To answer the two RQs we evaluate the ATD in terms of the AS and the Architectural Debt Index (ADI) computed through Arcan. Since we have large experience [21][210][79] in analyzing the ATD in open source projects, but not in MAS development platforms, through the answer to these RQs we aim to analyze the ATD of MAS platforms, in order to provide some preliminary hints to their developers. In case

8 A publication was extracted from this study [196], done in collaboration with Daniela Briola.

ATD is present or specific AS are identified in the systems, developers have to pay attention to these problems to prevent them or remove them as soon as possible.

ANALYZED PROJECTS We selected four well-known MAS development platforms, namely Jade [33], Jadex[40], Jason[39] and Netlogo[259], and analysed their development history. These projects are written in Java, the programming language supported by Arcan. All projects but Jade are hosted on Github, which, given the large amount of code commits (code snapshots at specific points in time), enables the easy analysis of their history. Table 6.7 shows the main project characteristics: names, the number of analysed commits, the considered time period (date of the first and last commit), size expressed in Number of Lines of Code (LOC) both for the first and last commit and finally the download url. Concerning the commit analysis, we considered only commits pushed or merged into the master branch, starting from the beginning of the commit history and by sampling one commit every 30 days. We conducted a different analysis for Jade, which is the only project not hosted on Github. We collected six versions from the Maven Central Repository[9] and run Arcan on all of them. Since we found only the jar files, we could not report the number of Lines of Code in Table 6.7.

Notice that the selected projects have different history, community, development team and purpose: indeed, different amounts of commits are sampled in each case. This makes the individual results difficult to compare directly. However, we propose a preliminary analysis which shall be complemented with manual validation from developers and additional context information.

COLLECTED DATA We collected data about CD and HL, both at class and package level, and UD. We also computed for each analysed version its ADI value.

We ran Arcan on the commits of each considered project and organized the results in a dataset, where each observation corresponds to a single commit of a single project. The columns of the dataset store the data about *1)* the project the commit belongs to *2)* the number of AS detected in the commit (one column for each type) and *3)* the value of the ADI of the commit. The dataset and the analysis script are available in the replication package[10].

ANALYSIS In order to answer our research questions, we conducted the same analysis adopted in the study about IoT ATD (see Section 5.2.2), i.e., statistical metrics computation (about ADI and AS), trend analysis and manual validation of the commit comments. No-

9 https://mvnrepository.com/artifact/com.tilab.jade/jade
10 https://gitlab.com/essere.lab.public/mas-atd-evaluation

Table 5.10: Projects characteristics

Project	#Commits	First commit	Last commit	LOC first commit	LOC last commit	Download url
Jade	6	23/12/2015	06/06/2017	-	-	https://mvnrepository.com/artifact/com.tilab.jade/jade
Jadex	111	05/11/2008	16/03/2018	130288	502220	https://github.com/actoron/jadex
Jason	38	23/03/2017	20/04/2021	37447	45825	https://github.com/jason-lang/jason
Netlogo	25	05/08/2011	09/05/2016	60260	56075	https://github.com/NetLogo/NetLogo

tice that, concerning trend analysis, the Mann-Kendall test can be used to find trends for as few as four samples. In our case, one sample corresponds to one commit. However, with only a few analysed samples, as in the case of Jade (only 6 versions), the test has a high probability of not finding a trend when one would be present if more commits were provided. Hence, we report also the results of Jade trend analysis, but knowing that they could be less relevant with respect to the other analysed projects.

FINDINGS We now report the results of our analysis and also the answers to our research questions. Table 5.11 reports results of the distribution analysis conducted on the four projects. The statistics are evaluated on the number of AS, also divided by AS type (CD, HL, UD), and on ADI, measured for each commit during the considered time period. The project with the highest mean number of AS is Jade ($\approx$ 879), and it has also the highest mean value of ADI ($\approx$ 38).

On the other hand, Netlogo has the lowest AS and ADI mean values. On the other hand, Netlogo has the lowest AS and ADI mean values. Notice that it is reasonable to have a non-zero number of AS in large projects as the considered ones. We analysed many Open Source Java projects in past works, indeed the ADI value is tuned with a reference dataset of past analysed projects. However, to be able to define how much is a "good" amount of AS in MAS platform is not a trivial task, because the answer is largely bounded to the development context (e.g., developers, developers skills, MAS platforms peculiarity). That is why in this study we mainly focused on the evolution of ADI and in grasping some insights about why the AS appear/disappear.

We can provide the answer to the first RQ:

Table 5.11: Distribution analysis results

Metric	Jade	Jadex	Jason	Netlogo
CD mean	846.83	123.38	61.95	9.06
CD std.dev	7.96	55.78	13.22	4.31
CD min	837	1	48	1
CD max	856	193	91	16
HL mean	5	1.62	3.34	1.00
HL std.dev	0	0.49	0.58	NA
HL min	5	1	2	1
HL max	5	2	4	1
UD mean	28	67.96	8.55	1.79
UD std.dev	1.67	28.19	2.36	0.43
UD min	27	1	7	1
UD max	31	103	15	2
AS mean	879.83	192.12	73.84	7.2
AS std.dev	7.22	83.32	13.71	6.09
AS min	869	2	59	1
AS max	888	289	110	18
ADI mean	38.33	10.80	23.45	3.96
ADI std. Dev	1.97	5.95	3.06	3.52
ADI min	35	3	19	0
ADI max	41	23	30	11

RQ1: Which is the most present type of AS in MultiAgents Systems platforms? The most present type of AS (on average) is CD. The less present AS is HL.

We also ran the Mann-Kendall tests to analyse the trend of the same variables (CD, HL, UD, AS and ADI). Table 5.12 reports the results only of the significant cases, i.e., with p-value < 0.05. The table also indicates whether the *trend* is increasing ($+$) or decreasing ($-$).

We now put in relation the results of the two analysis and provide a brief discussion of the architectural debt of each project. In particular, we manually checked the commit comments of each project, with a focus on the commits which presented large drops of the ADI value (points of interest). Our aim was to find a relationship between the change in the value of ADI and the content of the commit under analysis, starting from the description reported in the commit comment by the developers. For instance, if a sudden decrease in the ADI is

Table 5.12: Mann - Kendall test results

Project	P-value	Variable	Trend
Jade	0.019	ADI	-
Jadex	0.000	ADI	+
Jadex	0.000	AS	+
Jadex	0.000	CD	+
Jadex	0.043	HL	+
Jadex	0.043	UD	+
Jason	0.003	AS	+
Jason	0.000	CD	+
Netlogo	0.000	ADI	-
Netlogo	0.000	AS	-

backed by a comment stating that a major refactoring was applied in the commit, then the Arcan result is validated and we obtain an insight about practices for the removal of ATD.

The following figures depict the ADI trend (y-axis) of the projects, computed for each commit (x-axis). Table 5.13 reports the main points of interest in the projects commit history, identified by the *Date* of the commit, the *Commit hash*, the *ADI* value and the interesting *Characteristics* of the commit. The table does not report results concerning Jade because we conducted a different kind of analysis on it. Given that Jade is not hosted on Github, we could not analyse the commit comments, however we manually checked its changelogs.

JADE As underlined before, the scarce number of analysed versions may have hindered the trend analysis results. However, the Mann-Kendall test gave an output for the ADI variable. In particular, the ADI trend is decreasing, but not dramatically (see Figure 5.7). The detected ADI value ranges from 35 (last analysed version, 4.5.0) to 41 (first analysed version, 4.3.0). Given the few versions, we were able to manually analyse the changelog[11] of all of them. We checked for key-terms, namely *Improvement(s)* and *Fix(es)*. We noticed that each version is characterised by many fixes, with version 4.4.0 having the greatest number of changelog comments addressing them (8). Concerning improvements, we identified few of them (approximately one per version), with most of them referring to enhancements to security. However, version 4.5.0 reports a comment about "Improved code style and logging". A clean code style can improve maintainability,

11 https://jade.tilab.com/doc/ChangeLog, accessed October 2021

Table 5.13: List of ADI points of interest

Project	Date	Commit hash	ADI	Characteristics
Jadex	08/06/2009	09681d4371f53a1822de0a16c5b86e8349ea43c1	3	Min ADI value
Jadex	08/10/2009	5d266d02e4e9d90bde2dd942a7aff456eeca1aa4	3	Min ADI value
Jadex	14/12/2010	81be90b42d44a135e74d8a00213406951b78acaa	3	Min ADI value
Jadex	15/08/2011	a5b92b3f35a33222cd501bb37a2200e67d24187a	3	Min ADI value
Jadex	09/09/2014	fc28f548505583bfb2f1adcb1d3e368ec81aafd0	21	ADI drop, preceded by fixes and introduction of new data structure
Jadex	13/11/2017	6292d0cc21c24fa16627725e1bd0bfab52531222	9	ADI drop, preceded by fixes
Jason	20/04/2021	680921bbe8ff0247427d22e57ea3e36497143cd5	25	ADI drop, preceded by the implementation of a new test framework
Netlogo	07/02/2012	15daf0d82f11acc3a66ac9ca369fccf4efe77776	8	ADI drop
Netlogo	08/05/2012	5c5f707059b9a5c6546702cd2092b58e971b2632	6	ADI drop
Netlogo	17/05/2013	00ab7fae6c16c7a9b7e6b928080b15e0cd532e29	3	ADI drop
Netlogo	17/06/2013	82673cd6eda0f19b1dd533bd0e3a629ea24f23e6	3	ADI drop
Netlogo	31/01/2014	05710fe041397fd70286bc2347343993dd4f5563	0	ADI drop, corresponding to pull-request
Netlogo	13/03/2014	9056a8d98b69a2a984580d2cafceffc50685acb6	0	ADI drop, corresponding to pull-request
Netlogo	15/05/2014	c6a5902697212fb7adc14ae8d1e8a3e428e4dae8	2	ADI drop, corresponding to the addition of a new Scala submodule
Netlogo	04/09/2014	895609613fc1b5f592ff9eb84dcc5767f40ee7ec	0	ADI drop, corresponding to pull-request

and this could be reason behind the ADI value of this version, the lowest detected.

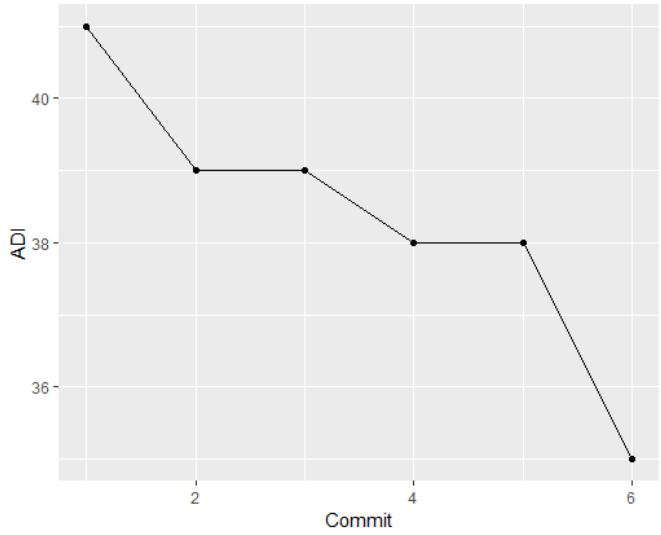


Figure 5.7: Evolution of ADI value - Jade

JADEX The ADI trend is increasing (see Figure 5.8). The same happens for all the other variables (number of AS, CD, HL and UD). Indeed, Jadex is the project with the highest mean number of AS.

We manually analysed the points in time where ADI reached its lowest values, with the aim to understand whether interesting practices to manage architectural debt could emerge. In particular, we analysed the five commits corresponding to the lowest values of ADI, equals to 3 (see Table 5.13). Unfortunately, there are no messages or comments associated to those specific commits. The only interesting aspect is that all the five commits were created by the same two authors. We also analyse the period of time between and 09/09/2014 comments report multiple time the word "fix" and also the adoption of a dedicated info structure for Non-Functional properties (`NFPropertyInfo` class). which are Non-functional property annotation.

Another point of interest in the Jadex history is on date 13/11/2017, when ADI drops from value 17 to 9. We checked the commit comments between the two points, corresponding to the changes made in a month, and all of them concern fixes. Some examples: *"**Fix** proxy factory class loader issue and component spec as class."*; *"**Fixed** most test failures caused by "config cleanup" commits"*; *"**Fix** component/bootstrap factory stored as string and as class"*.

JASON This projects ADI does not show any trend, as can be seen by the irregular shape of Figure 5.9). However, the number of AS and CD has an increasing trend. We manually analysed a sampled period, which comprises the commits between 17/08/2020 and 20/04/2021.

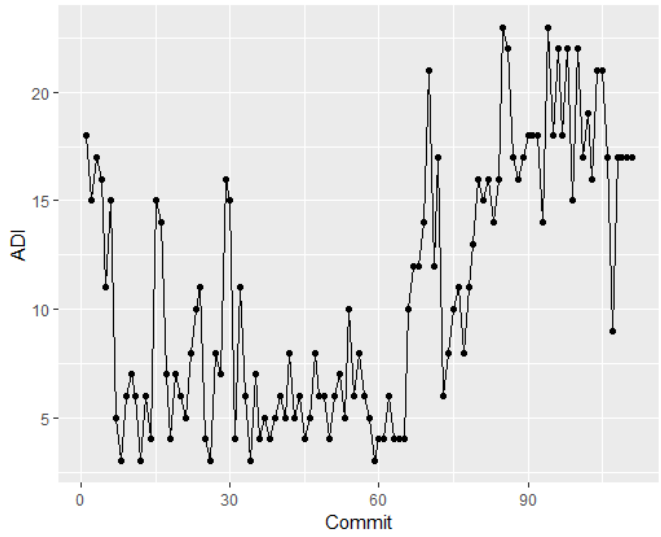


Figure 5.8: Evolution of ADI value - Jadex

First we analysed the period from the high peak (ADI=30) to the lowest point (ADI=19). From the commit comments and the changelog of the nearest release, it appears that the most meaningful development was the implementation of a new tests framework. Even if it is affected by less AS with respect to Jadex, Jason is the project with the highest average ADI value. This means that compared to Jadex, its AS are more critical (have highest severity [210]).

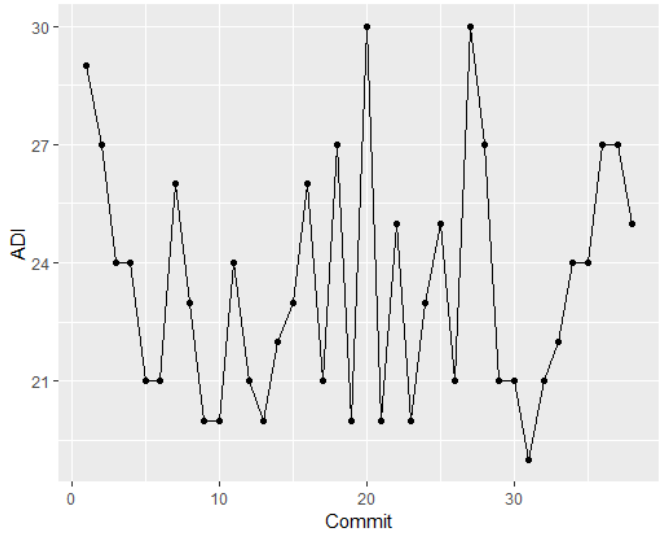


Figure 5.9: Evolution of ADI value - Jason

NETLOGO NetLogo ADI trend is the only one decreasing (see Figure 5.10). At the same time, the number of AS has a decreasing trend: we can deduce that the decrease of the value of ADI is not due to the decreased severity of the smells, but only due to the decrease of

the total number of smells. In general, Netlogo is the projects less affected by architectural smells and with the lowest values of ADI (see Table 5.11).

We manually analysed the commit history of this project, in particular we focus on the points where ADI decreases (see Table 5.13). Most of the associated commit messages indicate improvements: *"**Minor improvement** to Client Perspective Example."*; *"**Mostly-irrelevant correction** to a HubNet method's Scala style"*. However, there are no signs of big, structural changes which could explain the significant drops of the ADI value, apart from the presence of three pull-requests, corresponding to $ADI = 0$ and the introduction of a new Scala submodule providing network analysis tools for use in NetLogo (commit message: *"Add new network extension submodule!"*).

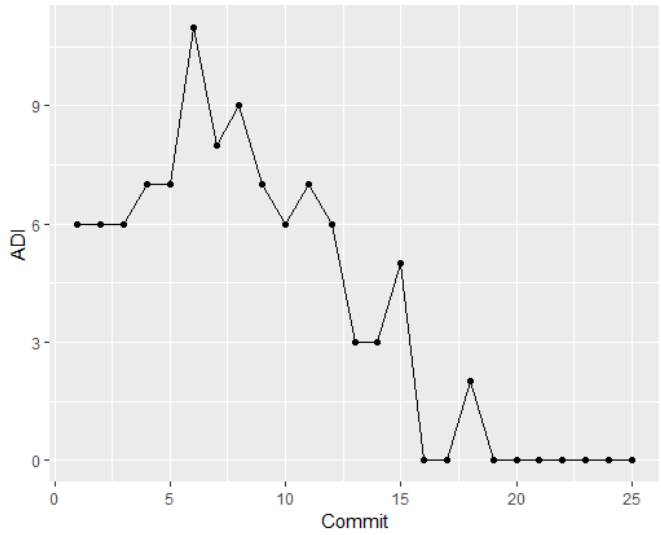


Figure 5.10: Evolution of ADI value - Netlogo

RQ2: What can we observe according to architectural debt of the considered MultiAgents Systems platforms? All the analysed projects present architectural debt along their development history, but with different trends. Jadex has an increasing ADI, while Jade and Netlogo show a descreasing trend, with Netlogo having the last commits with zero debt. Jason did not present any trend.

THREATS TO VALIDITY The threats to validity of this study are the same of the previous one about IoT platforms (Section 5.2.2).

FINAL REMARKS We exploited our tool Arcan to analyse four Open-Source MultiAgent Systems (MAS) development platforms and we evaluated their Architectural Technical Debt (ATD). We investigated the outcome of the tool by manually analysing the commit comments

available on Github, for three of the four projects, and the changelogs for one of them. From our analysis, we acknowledged that the considered MAS platforms are affected by architectural debt, in particular Jade is the most affected, while Netlogo is the less affected, with a decreasing ADI trend.

From the manual analysis, we could not find clear indication of practices to manage architectural debt. However, for all the projects, in the points in time where ADI reaches its minimum, the comments refer to "Bug fixing", "Improvements" or pull requests. This could mean that architectural debt, usually considered only at architectural level, has also a relationship with issues at code level, such as bugs.

5.2.4 *Sen4Smells: A tool for ranking architecture-sensitive smells for a debt index*

As outlined in the introduction to the chapter, the quality of a software system can be evaluated by considering the technical debt accumulated in the system [123]. To this end, Arcan is not the only tool offering a Technical Debt Index (TDI): other software analysis tools such as Sonargraph, CAST and Sonarqube (see Section 2.5) offer their own index [20] [26]. Once developers have chosen a TDI and have evaluated it on their project, a relevant aspect for them is the "intepretation" of the index values [155]. By interpretation, we refer to the ability of examining the index values in order to spot those AS (or other system elements) that are the main *contributors* to the current debt. This task can either involve looking at system elements with different *granularity* (e.g., packages, classes, types of smells), or considering the history of a system element (i.e., the system evolution). Common questions posed by engineers include: (i) which packages are the most sensitivity ones for the current architecture health?, or (ii) which smells have suffered instabilities in the past system versions that might compromise the design in future versions? As a result, the AS or packages being sensitive for the system architecture should be reported, due to their impact on system evolution. Unfortunately, the examination of TDI values is often a cumbersome and time-consuming task for engineers.

In this context, we developed a tool called *Sen4Smells*[12] that performs an automated Sensitivity Analysis (SA) for a collection of system values provided by a predetermined TDI. Our approach relies on two building blocks: (i) the adaptation of an existing SA method to debt indexes based on AS, and (ii) a decomposition strategy for the index at different levels. At the lowest level, we leverage on AS and TDI metrics (or features) associated to those smells. The goal is to assess how TDI variations can be attributed to variations in features of system elements [54]. To do so, the SA performs a screening of the various system elements affecting the index over time, and returns a ranking with the most sensitive ones to the tool user. The inputs for this analysis are: a list of previous system versions, a particular TDI, and the desired granularity of system elements. *Sen4Smells* is designed as a pipeline, in which existing modules for detecting AS and computing metrics from the source code can be wrapped, and the user can configure them based on the selected TDI.

The main aim of this tool is the assistance for developers to interpret TDI values in terms of problematic AS and system packages, as *indicators* of system quality trends. We do not develop a new SA technique, but rather select a suitable one and adapt it to our index

12 A publication was extracted from this study [50], done in collaboration with Antonela Tommasel and Andres Diaz Pace.

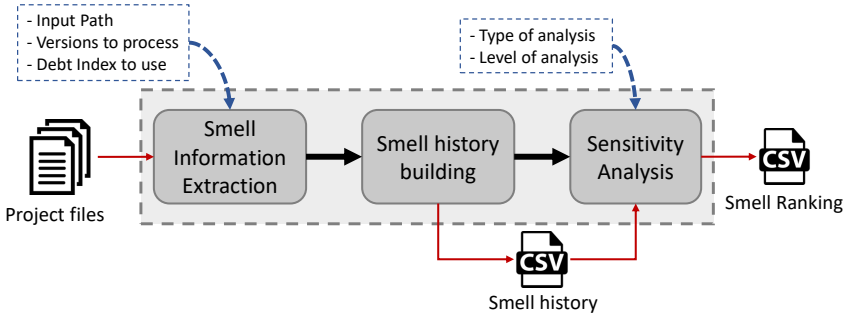


Figure 5.11: Main processing stages and parameters of *Sen4Smells*.

interpretation problem. To evaluate the *Sen4Smells* functionality, we have instantiated the pipeline with two indexes: our ADI and SDI (Structural Debt Index), provided by the Sonargraph commercial tool (see Section 2.5). Sonargraph computes cycles and metrics on Java code. The SA is currently implemented via the well-known Sobol method [214]. Other TDI formulas, smell analyzers, or alternative SA methods can be integrated into the pipeline.

TOOL ARCHITECTURE & BACKGROUND *Sen4Smells* is designed as a 3-stage pipeline architecture, as shown in Figure 5.11. A prototype and examples are available[13].

The first stage, called *Smell Information Extraction*, processes a sequence of Java system versions for the project under analysis, in order to detect instances of AS and to compute features associated to these AS. The system versions (project files) to analyze are provided as input by the tool user. We assume she/he is looking for design problems at the current version (i.e., the latest provided version). The version processing is delegated to wrappers of existing tools (e.g., Arcan, Sonargraph). The user also specifies the debt index to be used. In our context, a TDI is based on predefined AS types and features. For example, Sonargraph Structural Debt Index (SDI) is a cumulative function of the cycles (i.e., the smell type) and the number of dependencies to be removed (e.g., a feature) to break those cycles.

The second stage, called *Smell History Building*, builds the evolution history of the different AS across the range of versions. This evolution history can be seen as a matrix [127], in which each column represents a version at time t and each row represents a smell instance. As each row is the combination of a smell and a feature, the cells of the matrix keep the values of the smell features across versions. In this way, we can trace "paths" of smell variations over time. Note that the user's focus is on AS appearing in the current version (i.e., the last known version), whose behavior can be explained through the history of the smell features in past versions. Once computed, the evolution history of the AS is stored in a CSV file to be processed by the SA. Figure

13 See at https://github.com/tommantonela/Sen4Smells

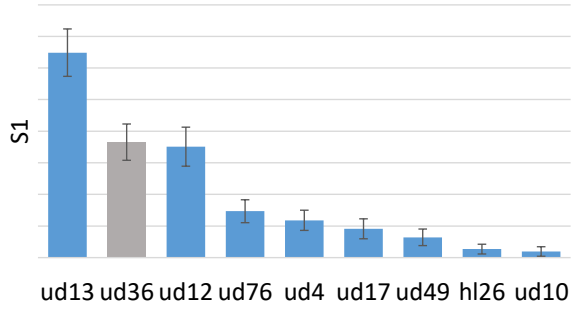
	openjpa 1.0.0	*openjpa 1.1.0*	*openjpa 1.2.0*	*openjpa 2.0.0*	*openjpa 2.1.0*	*openjpa 2.2.0*	*openjpa 2.3.0*	*openjpa 2.4.1*
ud13	1.85	4.1	2.05	4.5	2.25	2.3	2.3	2.25
ud36	1.89	2.28	2.56	1.89	1.62	1.35	1.62	0.6
ud12	1.8	2.05	2.1	2.25	4.7	4.8	2.4	2.35
ud76	0.6	1.7	1.7	2.43	1.8	2.43	1.8	2.43
ud4	0.81	0.81	1.26	1.26	1.8	1.8	1.8	2.43
ud17	0.68	0.68	0.36	1.66	1.57	1.42	1.42	1.66
ud49	0.8	0.8	0.8	1	1.53	1.53	1.53	1.53
hl26	0.35	0.5	0.5	0.45	0.5	0.55	1.2	1.2
ud10	0.48	0.13	0.13	0.21	0.28	0.21	0.21	0.28
cd34	-	-	-	-	1.05	0.9	0.9	1.05
cd35	-	0.27	0.2	0.27	0.27	0.3	0.3	0.3
cd48	1.26	1.26	1.26	1.59	1.59	1.59	1.59	1.59
ud7	-	-	-	0.26	0.26	-	-	0.26

Figure 5.12: Evolution of scores for smells across different OpenJPA versions.

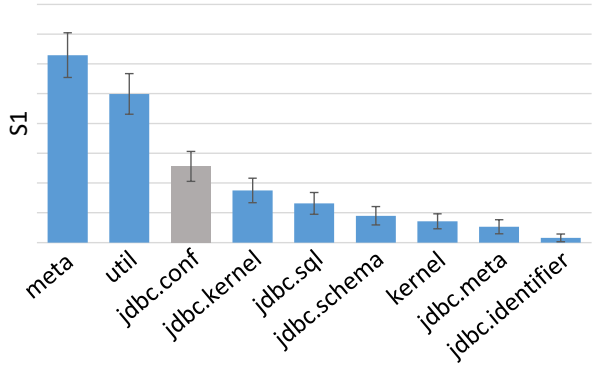
5.12 shows a matrix with the evolution of index scores for a subset of AS (left-most column) detected in 8 versions of Apache OpenJPA.

The third stage, the last one, is the *Sensitivity Analysis*, which takes as input the smell evolution paths and the chosen TDI to report a ranking of smells (or other system elements) to the user. Given a TDI formulation (e.g., the ADI definition, Section 5.1), we see it as a black-box model that relates inputs (the values of the AS features) to a numeric output (the index value for each system version). This model is exercised with the Sobol method, which is a global SA method for measuring the contribution of the inputs to the output variance (other SA methods are also possible). As a result, a sensitivity score (also known as Sobol index) is assigned to each AS. The higher the smell score, the higher the chance that variations in the TDI are due to that smell. Thus, we interpret the score as the architecture sensitiveness of each smell regarding the whole system design. The SA can be performed at the level of AS (which is a fine-grained level), but also at other granularity levels. For instance, the user can be interested in grouping the AS according to the system packages or smell types. In the former case, the smell features are grouped per top-level package, and the SA returns a ranking of critical packages. In terms of the tool concepts, this means that the TDI can be decomposed using different criteria, and then the SA is adjusted accordingly. For example, Figure 5.13 shows a report of key AS and packages for OpenJPA (X axes) using the Sobol method.

INDEX DECOMPOSITION STRATEGY For analyzing TDI values, we propose a decomposition of the (global) index formula into its constituent parts. These parts (or elements) can refer to different granularity levels, namely: (i) the AS themselves (bottom level), or (ii) groups of AS using a predefined criterion. For example, we might group the AS based on either their type or the package structure of the system. At each level, the corresponding elements are characterized by the features of the index formula, which altogether provide index values for the elements. This decomposition strategy is sketched



(a) Sobol Indices for key smells



(b) Sobol Indices for key packages

Figure 5.13: Results of sensitivity analysis for OpenJPA

in Figure 5.14, with an example of an intermediate level of AS grouping.

From a temporal perspective, our strategy looks at the index values over a range of system versions, independently of the granularity level under consideration. This way, we can analyze trends in the evolution of AS, or in the evolution of packages (as groups of AS). Departing from the index decomposition, the elements at each level are seen as variables whose values follow a particular distribution over time. These distributions are fed into the SA for interpreting the TDI.

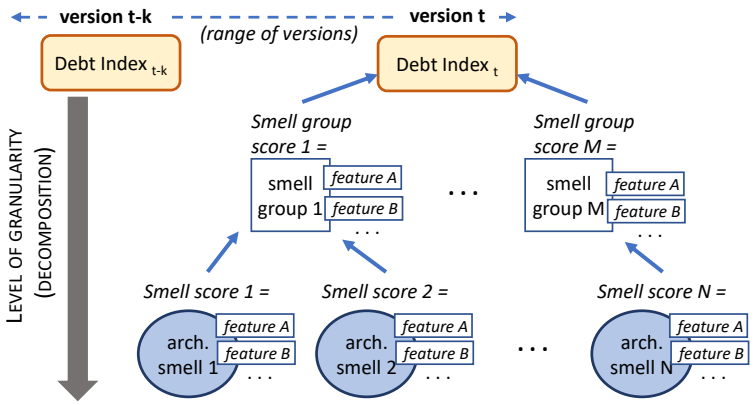


Figure 5.14: Decomposing a debt index in granularity levels and over time.

SENSITIVITY ANALYSIS SA techniques study how the variation in the output of a model can be apportioned among model inputs [214]. In our problem, the input variables are the AS features or index scores (at a given decomposition level), while the model output is the global index value. If a change in a variable results in a relatively large change in the index, then we say that the index is sensitive to that variable. From this analysis, a ranking of key variables (e.g., AS or system elements) can be obtained. For instance, for OpenJPA in Figure 5.13, the SA employed a model with around 90 variables for AS. In general, the selection of a SA method depends on the model characteristics, the computational efforts to run the model, and the SA objectives, among other factors.

The *Sobol* method [214] is a global SA technique, which allocates the output variability to the variability of the inputs taking into account all the variables and interactions among them. The results of this method are the so-called *Sobol indices* for the input variables. The higher the value of a sensitivity index, the more influential the respective variable is for the model. First-order indices (S1) reflect the main effect and measure the fractional contribution of a single variable to the output variance. Total-order indices (ST) take into account the main, second-order and also higher-order effects of variables on the output variance. Figure 5.13a shows the S1 values for the top-10 AS for the ADI. For instance, an S1 of 0.18 was obtained for `ud36`, which indicates that this smell is very likely to influence the index. A plausible developer's interpretation of the situation is that a few UD smells are more problematic (in terms of increased dependencies) than a large group of CD smells (if their cyclic dependencies remain stable). If we instead analyze the package `org.apache.openjpa.conf` in Figure 5.13b, an S1 value of 0.12 reflects an influential role of (the AS in) the package for the ADI. The analysis reveals other influential packages as well, such as: `org.apache.openjpa.meta` and `org.apache.openjpa.util`.

The transformation of index features into SA variables involves mainly three tasks: (i) specifying the decomposition level for the variables (e.g., smells, packages), (ii) sampling values from the evolution history of those variables, and (iii) computing the Sobol indices. For sampling the values of the variables, we rely on their distribution in a range of system versions. For CD smells, the distribution often shows cycles becoming larger (or smaller) over time; while for UD and HL smells the variations are due to an increase (or decrease) in the number of dependencies to the packages affected by the AS. If AS are grouped, an analogous distribution can be derived.

CASE-STUDIES Currently, *Sem4Smells* is able to work with the ADI (Arcan) and the SDI (Sonargraph) indexes. Each TDI needs an integration with the corresponding tool, which is adapted and parameterized in the pipeline. A video with usage examples for both cases

is available[14]. The whole pipeline is executed from the command line using different configuration options.

ARCAN & THE ARCHITECTURE DEBT INDEX The *Smell Information Extraction* component takes the JAR files of the system versions, and passes them on to a wrapper for the Arcan engine. When invoking Arcan on a given version, it generates a "version" object that contains all the detected AS and their ADI features. To ensure extensibility, this version structure is independent of the particular TDI. Then, the *Smell History Building* component merges all versions into a smell evolution matrix, which is saved to a standard CSV file.

The *Sensitivity Analysis* component reads from the matrix to run the Sobol analysis. The desired granularity level for the analysis is configured in this component, which triggers the creation of AS groups as Sobol variables. This functionality is open to the integration with any index or AS type. In addition, the TDI formula that maps the variables to index values needs to be set. For ADI, this formula was implemented via a general class of the component. A CSV file for each ranking is finally outputted.

SONARGRAPH & THE STRUCTURAL DEBT INDEX Unlike Arcan, each system version needs to be initially processed by Sonargraph, and then exported as an XML report file. This file contains information about cycles and SDI features. In the *Smell Information Extraction* component, we rely on a Sonargraph API[15] for accessing each XML file as a version object. The process continues with the generation of the smell evolution file by the *Smell History Building* component. Finally, the *Sensitivity Analysis* component runs the Sobol method as in the Arcan case, except that the SDI formula is configured as the index to be used.

FINAL REMARKS In this section, we have described the *Sen4Smells* tool that performs a sensitivity analysis of a TDI based on the underlying smells in the formula. A direct benefit of this analysis is that makes a TDI actionable for engineers, by enabling the identification of key AS and problematic packages. As future work, we plan to integrate the tool pipeline within a build process of a project. Also, visualizations and reports based on the SA rankings could improve the tool.

Regarding the Sobol method for the SA, we found it useful because it makes no assumptions about the index formulations. However, from experiments with projects with a large number of smells, we observed that the computational efforts required by Sobol might increase rapidly with the number of variables. In such cases, more efficient methods should be explored. A possible related study is to

14 See at https://www.youtube.com/watch?v=6RL0qCqZYPM

15 https://github.com/sonargraph/sonargraph-integration-access, accessed October 2021

validate whether the rankings are correlated with critical parts of the analyzed systems, where the criticality can be measured with metrics such as PageRank (see Section 5.3) or can be assigned to each part of code by the developers working on the project under analysis.

5.3 AS CRITICALITY EVALUATION

When describing our Architectural Debt Index (ADI), we made reference to the computation of the AS *PageRank* and *Severity*. They are metrics useful to evaluate specific properties of AS, namely criticality and cost-solving (see Section 2.2.2).

Criticality of an AS models the degree of removal urgency associated to the AS, i.e., the smell should be removed as soon as possible because it affects a part of the project which is important for the developers (e.g., frequently changed or highly referenced) or has a strong impact on the maintainability of the project. On the other hand, *cost-solving* (cost of fixing, cost of refactoring) of AS is the effort needed to remove a smell from the system [209]. This variable depends less from the perception of the developers but more from the specific characteristics of the interested AS. To resume, during AS management, developers can take into consideration two distinct aspects concerning smells: their criticality, i.e., how much is important to remove them as soon as possible (urgency), and their cost-solving, i.e., how much it costs to remove them.

In this section[16] we describe our work regarding the investigation of the relationship between criticality and cost-solving, measured with the Severity and PageRank metrics: see Section 2.3 for the definition of the Severity of each AS, see Formula 12 in Section 5.1 for PageRank definition. We previously conducted a preliminary study in which we graphically analysed the trend of Severity and PageRank and also started investigating their possible correlation [82]. We took in consideration six single-version projects. We then extended the study by conducting an empirical evaluation on a total of 264 versions of 10 projects with the aim to empirically study criticality and cost-solving during the evolution of the projects, and investigate whether there is a correlation between the trends of the two metrics. The following Section reports the results of the analysis.

5.3.1 *Empirical Study Design*

The study aims to answer the following Research Questions (RQ):

- *RQ1: How PageRank and Severity of the smells evolve in the version history of a project?*
- *RQ2: Can we find some correlation between PageRank and Severity by considering each type of smell?*

The answer to *RQ1* aims to analyze if the values of the two metrics tendbut to increase or decrease in the version history of the projects. Moreover, we are interested in understanding which AS type(s) tend

16 A publication was extracted from this study [200]

to become more critical and/or difficult to remove in the version history of a project, where the criticality is evaluated through the PageRank and the cost solving is estimated with the Severity metric. In this way a developer can decide to focus the attention on these types of smells first.

The answer to *RQ2* allows to evaluate the correlation between the criticality and the cost solving of a smell. If for example the values tend to go together, highly correlated, for a specific type of AS, it means that as long as the smell is critical, it is also hard to remove and vice-versa: in this case, the two metrics would produce the same ranking of smells, i.e., the prioritization of the smells would be equal by considering one of the two metrics interchangeably.

In case of positive correlation, it could be also in any case interesting to analyze possible outliers with different values of the metrics (high/low) and better capture the relevance of the metrics. We could find that the two metrics have a strong positive correlation for a specific type of smell, and not for other smells. This scenario can outline the relevance of the metrics for each type of smell. Otherwise, no correlation, we could infer that there is no link between the urgency of removing a smell and the cost of removing a smell, as computed by the proposed metrics. In this case a developer can decide to not remove an AS with low PageRank and high cost solving, and to remove first an AS with high PageRank and low cost solving, since this AS could become more critical since it appears in a central part of the project.

We aim with our study to provide developers insights on the evaluation of criticality and cost solving of AS through the PageRank and Severity metrics. Severity metric is focused on evaluating the cost solving in terms of the number of project dependencies affected by the smells, while PageRank is more focused on the importance (criticality) of the affected components (classes/packages). Hence, both metrics could be useful to determine the prioritization of AS, i.e., help the developer in choosing which smell to refactor first depending on the developer's needs, i.e., the need to address the most critical ones first or the most expensive ones.

Since, as already outlined, we exploited the two metrics to compute the ADI value, the results of this study can be useful also to evaluate whether the two metrics truly capture different aspects of a smell or not. In the latter case, one of the two metrics could be left out.

We describe below the analyzed projects, the data we collected on AS, their Severity and PageRank and the data preparation and analysis.

ANALYZED PROJECTS We analyzed several versions of 10 projects, for a total of 264 versions (see Table 5.14). Most of the chosen projects were picked from the Qualitas Corpus [239]. We selected these projects

Table 5.14: Summary of the dataset

Project	#V	#CD-Cl	#CD-Pkg	#HL-Cl	#HL-Pkg	#UD	#AS
Ant	24	8131	2064	15	92	243	10545
Azureus	24	97172	29801	41	70	3478	130562
FreeCol	24	30488	1652	86	54	356	32636
Hibernate	24	12910	9026	18	129	1267	23350
JMeter	26	3930	2681	79	54	574	7318
JGraph	24	2602	79	79	1	51	2812
Jstock	24	13585	619	64	8	247	14523
Jung	22	894	658	31	27	270	1880
Lucene	31	6241	407	9	59	187	6903
Weka	44	25241	5200	102	41	1042	31626

Acronyms. V: version, CD: Cyclic Dependency, HL: Hub-Like Dependency, UD: Unstable Dependency

since they have already been the subject of several studies, they are publicly available and enable the replication of this study. These data were also combined with data from the MavenRepository[17], also publicly available. We considered several releases for each project.

To easily compare the different projects, we chose roughly the same amount of versions and preferred different releases, major or minor, over patches when possible. In general, in this paper we use the term version to refer both minors and majors. The chosen systems also vary in size and number of smells (see Table 5.14). In the column group *last version* we report the projects' size (in terms of classes/packages) and number of AS of the *last version* of the project in the development history.

DATA COLLECTION AND ANALYSIS We performed this study by considering three (HL, CD, UD) of the six AS detected with Arcan, but also the other AS can be considered in the future. We limited the analysis on the following three smells since they are the only ones for which we integrated the Severity metrics into the definition of ADI. We ran Arcan and we pre-processed the output data in order to produce the dataset for our analysis. Other than Arcan, we exploited the Knime platform[114] and R programming language [241] for the processing and statistical analysis of the data. The resulting dataset is a collection of 262155 smells categorized by project, version, type, granularity level, Severity and PageRank. Table 5.14 shows the summary of our dataset, where we report the project size and the number of smell instances, divided by type: for each project (considering *all versions* in history) we show the number of detected CD at class and

17 https://mvnrepository.com/

package level (*CD-Cl* and *CD-Pkg*), of detected HL at class and package level (*HL-C* and *HL-P*), of detected UD (*UD*) and the sum of all project's AS (*AS*). A *smell instance* corresponds to one occurrence of the smell in the project, thus the reported numbers are the counts of all the occurrences.

We studied two different aspects: 1) Severity and PageRank evolution, in order to answer RQ1; 2) Severity and PageRank correlation to answer RQ2.

Concerning evolution, we analyzed the evolution of the two metrics for each type of smell in order to study their different behaviours. We summarised the data for each version by averaging the values of both metrics with respect to the total number of smells detected in the version. We conducted trend analysis to understand how the average values of PageRank and the different types of Severity evolve overtime. We exploited the *Mann-Kendall test* (see Section 4.1).

We also analyzed the two metrics' evolution with respect to the evolution of the projects' size, where size corresponds to the number of classes and packages of the projects under analysis, to check whether the two things are correlated. We ran Spearman and Kendall correlation tests to investigate this aspect.

Concerning the correlation analysis of PageRank and Severity, we first tested the normality of our data. Given the large size of our dataset, we used Q-Q plots [260] to evaluate if the measures do not follow a normal distribution. A Q-Q plot is a graphical method for comparing two probability distributions by plotting their quantiles against each other. These plots are often used when the dataset is large enough to introduce bias in the Shapiro-Wilk test [219], which is a commonly used normality test. The Q-Q plots of all the projects showed a non-normal behaviour. Then, we tested the correlation between Severity and PageRank for each version of the projects. We computed the correlation on the metrics data of all smell type together and also separately for each smell type. We also computed the correlation separately for each granularity level, to contextualize the results at package or class level. Given the non-normal distribution of our data, we chose the Spearman's [227] and Kendall's [119] coefficients to calculate the correlation.

5.3.2 *Results*

We report the results both for PageRank and Severity evolution and their correlation. At the end of each section, we also report the answer to the relative RQs. All the results and plots can be found in the replication package[18].

18 https://figshare.com/articles/dataset/_/13636472

Table 5.15: Mann-Kendall results - PageRank

Project	Trend	P-value	Reference AS
Ant	+	0.009867	CD-package
Azureus	+	2.77E-05	CD-class
Azureus	+	0	CD-package
Azureus	+	3.81E-06	HL-class
Azureus	+	0	HL-package
Azureus	+	0	UD-package
Hibernate	+	0.030929	CD-class
Hibernate	+	0	CD-package
Hibernate	+	0.000677	HL-class
Hibernate	+	0	HL-package
Hibernate	+	2.38E-07	UD-package
Jgraph	+	0.001375	HL-class

EVOLUTION RESULTS In order to answer RQ1, we checked the trend of PageRank and Severity values throughout the versions of the projects. For every project and for both PageRank and Severity, we run the Mann-Kendall test. Table 5.15 and 5.16 show the outcome of the test, namely reporting the *Trend* (increasing + or decreasing -), the *P-value* and the *Reference AS* (the type of smell which the PageRank refers to) for PageRank, while *Granularity* (class or package) for Severity. The tables report only results where $p - value < 0.05$, i.e., there is a trend. We outline from Table 5.15 and 5.16 the following remarks:

- PageRank and Severity show a trend during time in few projects. We found PageRank trend in four over ten projects, while Severity showed a trend in five projects. The tables only show the projects with a positive or negative trend.
- Concerning the Severity of CDs, we observed both positive and negative trend at class level, in 4 projects, and a negative trend at package level, in one project.
- Concerning the Severity of HLs, we had examples at both class and package level of positive trends.
- The Severity metric of Unstable Dependency smell does not show a trend in any project, and we could notice only one project (Hibernate) where the PageRank of UD smells had a trend.

Table 5.16: Mann-Kendall results - Severity

Project	Trend	P-value	Granularity
Severity - CD			
Azureus	+	0.024848	class
Hibernate	+	0.000291	class
Jstock	-	0.025486	package
Jung	-	0.039728	class
Lucene	-	3.25E-06	class
Severity - HL			
Jstock	+	0.002832	package
Lucene	+	0.000422	class
Weka	+	0.002132	class
Weka	+	0.005923	package

We extended our analysis to see if the project size (measured by number of classes and packages) is correlated with the values of PageRank and Severity. We tested it for each project over its development evolution. We then analyzed the distribution of the correlation on the data of all projects. The first thing we noticed is that the number of classes and packages increases overtime.

However, this does not happen for Severity and PageRank values: we do not find a significant correlation between size and the metrics except for the correlation between PageRank computed on AS on packages and the number of packages in the system. The correlation values, computed for all the projects, have range in $[0.34, 0.89]$, with median equals to 0.74. We hypothesise that the correlation is high for PageRank because of how it is computed: the more the number of packages, the more the dependencies and higher the PageRank values are. For this reason, one may say that this should be true also for PageRank computed on classes correlated with the number of classes: instead, their correlation values range in $[-0.87, 0.9]$ with median equals to 0.45. This result may be due to the high variance in the number of classes among the projects (variance which is smaller for what concerns packages).

RQ1 Answer *How PageRank and Severity of the smells evolve in the version history of a project?*: in general we found that the average values of PageRank and Severity do not have a trend (neither positive or negative) over time.

Concerning the comparison with projects' size evolution, we found out that PageRank computed on packages show a posi-

Table 5.17: Severity and PageRank correlation (last version only)

Project	Version	Spearman	P-value	Kendall	P-value
Ant	1.10.7	0.582	< 0.001	0.46	< 0.001
Azureus	4.8.1.2	0.871	< 0.001	0.704	< 0.001
FreeCol	0.10.7	0.809	< 0.001	0.64	< 0.001
Hibernate	4.2.2	0.719	< 0.001	0.573	< 0.001
JMeter	5.2.1	0.575	< 0.001	0.455	< 0.001
JGraph	5.13.0.0	0.664	< 0.001	0.581	< 0.001
Jstock	1.0.6w	0.621	< 0.001	0.494	< 0.001
Jung	1.7.6	0.643	< 0.001	0.506	< 0.001
Lucene	4.3.0	0.411	< 0.001	0.33	< 0.001
Weka	3.7.9	0.53	< 0.001	0.428	< 0.001

tive correlation with the evolution of the number of packages: this is reasonable, since the increase/decrease in the number of packages has an impact also on the creation/deletion of package dependencies, thus on PageRank.

CORRELATION RESULTS In order to answer RQ2, we report in Table 5.17 the results of the correlation between Severity and PageRank, evaluated on all AS, not considering their type. As can be seen, the majority of the projects presented a strong positive correlation ($\rho > 0.6$).

Following, we discuss the correlation results by considering the different types of AS.

The coefficient values are bounded between:

- (**CDs**) 0.427 and 0.942 with Spearman's and between 0.214 and 0.812 with Kendall's;
- (**UDs**) 0.253 and 1 with Spearman's and between 0 and 1 with Kendall's;
- (**HLs**) -1 and 1 for both coefficients.

Due to their low occurrences, the metrics of HL and UD usually present a strong correlation. However, there are cases in some projects versions where the scarce number of detected smells makes this calculation misleading: in some cases correlations are very high, in other ones are very low (fluctuate). On the other hand, CD is the most common smell in the dataset and this has an effect on the correlation values: they largely vary in the dataset, making CD the smell type with some of the highest correlation values and at the same time the smell with some of the lowest correlation values.

However, a clear result is that for all projects the correlation at package level between PageRank and Severity of CD is strong, with the exception of JGraph (see the following paragraph).

OBSERVATIONS ON WEAK AND NEGATIVE CORRELATIONS From Table 5.17 we can observe that some projects, such as JMeter, Lucene, Weka and Ant show a weak correlation between the two metrics. We aim to investigate these behaviours and we start by analyzing two projects: JMeter, having a weak correlation, and JGraph, showing non-positive correlation values for CDs at package level. We focus on the last version of both projects because it is associated to the most updated codebase, hence we assume it is the most exemplary for them.

By analyzing the correlation coefficients of *JMeter*'s AS, we noticed that when they are calculated separately for each AS type, they present higher values than the ones reported in Table 5.17. Using Spearman's as an example: 0.575 is the ρ value by not considering the AS type and 0.638, 0.9, 0.881 are the values for CDs, HLs and UDs respectively. The values seem to imply that actually, while the correlation in general is weak for this project, when we look at the specific smell types, the two metrics tend to be positively correlated. However, the number of HLs and UDs in JMeter is very small compared to the number of CDs. Since correlations computed on few observations are not significant, we can conclude that only the correlation value computed on CDs is relevant for JMeter, and it explains why the overall correlation value is weak for this project.

If we closely analyze *JGraph* evolution, initially it shows a negative correlation for CDs at package level, which progressively increases (0.2 in version 5.10.0.1) and becomes strongly positive (0.73) in version 5.12.1.0. We further investigated what caused these changes in the correlation values. In the first versions with negative correlation we observed 3 CDs at package level, two of them with similar Severity and PageRank values and one with a strongly higher PageRank value, probably the cause of the negative correlation. After version 5.10.0.1 we noticed the presence of a 4th one. Its Severity was in line with the others and also its PagerRank: this likely balanced the PageRank values and subsequently caused the increase of the positive correlation. Hence we can conclude that the variations in the correlations values from negative to positive were due to the introduction of a new smell instance, whose metrics values strongly impacted the correlation values due to, as for JMeter, the general small amount of smell instances. However, this specific case does not represent a common behaviour in our dataset.

RQ2 Answer *Can we find some correlation between PageRank and Severity by considering each type of smell?*, we found out that the smell type showing the highest PageRank and Severity correla-

tion is CD at package level. However, also the other types, HL and UD, showed strong correlations, but given the lower amount of HL and UD instances, we consider the result regarding CDs more meaningful. We also investigated specific cases of projects with weak correlation and negative correlation but we did not find further insights.

5.3.3 *Discussion*

We found a strong correlation between PageRank and Severity. This means that, concerning the analysed data and the considered smells, the criticality and the cost-solving of smells go hand in hand: in the case of this study, if a smell affects an important (unimportant) part of the system, then it will also have a high (low) cost solving. We can outline two different interpretations of the results. The positive correlation could be due to the nature of the two metrics, both bounded to the dependencies of the system. In this case, the conclusion would be that PageRank and Severity capture the same characteristic of the smells, and one of the two is redundant. As consequence, in the ADI computation [17], only one of the two metrics should be used to evaluate AS criticality.

However, given how the metrics are defined, they differ one from the other. Severity takes into account the dependencies which are directly affected by the smell, while PageRank considers also dependencies outside the smell which converge towards the components affected by the smell. Figure 5.15 shows an example of two classes affected by the CD smell: the class on the left presents a high PageRank, due to the high number of incoming dependencies, and a low criticality; the class on the right has low PageRank, but since it is involved in two cycles, one of which is also large, its Severity is high. When considering PageRank, the most dangerous smell is represented by the first example, but when considering Severity the second example has the highest value. By combining the two metrics, both smells result crucial.

With such premise, the two metrics would capture different aspects of the smells, and their positive correlation could mean that critical parts of the system attract AS which are more expensive to solve.

Moreover, one could ask where is the difference in using PageRank when we could use simple coupling metrics such as FanIn and FanOut [150]. However, when evaluating the coupling of a component, such metrics take into account only the incoming or outgoing dependencies of the component itself. On the contrary, the PageRank value of a component takes into account the PageRank of all the components belonging to the dependency graph. In particular, the PageRank of a component is defined recursively and depends on the number of dependencies and the PageRank metric of all the components

that reference it (incoming dependencies). In this way, a component having many incoming dependencies but referenced by components with few incoming dependencies, is less important with respect to another component with many incoming dependencies and referenced by other components with many incoming dependencies. That is why PageRank is said to evaluate the importance of a component with respect to the entire graph.

From our analysis it results that the positive correlation is particularly evident in the case of CD. The reasons behind the CD Severity high correlation can be multiple: a part of code with high PageRank is interested by more changes [255] with respect to other parts of code, and thus more open to the introduction of (structurally complex) CDs. This is interesting because in the literature we find studies which confirm the correlation in the other direction [133], i.e., the presence of AS makes the components more prone to change: if our hypothesis can be further corroborated, the conclusion would be that the relationship between PageRank and CD Severity is like a dog chasing its tail, one triggers the other. Another reason could be that components with high PageRank are involved in a high number of dependencies, thus still making easier for a developer to wrongly introduce new entangled dependencies and create cycles very difficult to remove.

To conclude, there is a positive correlation between AS Severity and PageRank, however at the moment we cannot draw a definitive conclusion about how to interpret this finding. We plan to conduct a validation of our results with developers from industry, who could evaluate the ability of the two metrics to capture criticality and cost-solving, and also manually check the specific cases where smells have high PageRank and high Severity.

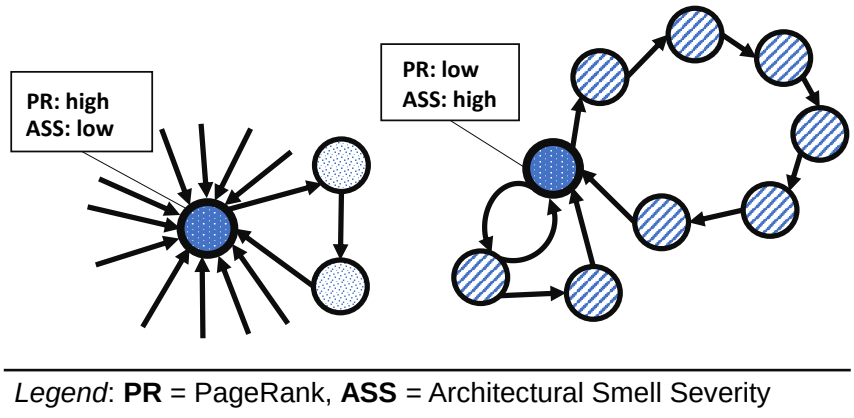


Figure 5.15: JUnit example of CD smells

5.3.4 *Threats to validity*

Our study presents some threats to validity which we address by following the structure suggested by Yin [264]. Concerning the *construct validity*, the two metrics, PageRank and Severity, may not measure what we claim they do, i.e., the criticality of the AS. However, this is a preliminary study and the next step is to validate the current definition of the metrics with developers, by letting them check whether the prioritization produced by the metrics is significant or not.

Other threats regarding the *internal validity* could be related to the choice of the statistical methods used for the correlation analysis and their implementation in the used tools, but we exploited very well known and used tools (R language). Moreover, we did not validate the two metrics by investigating the perception of developers of PageRank and Severity. However, PageRank was adopted in other studies as software ranking metric [192][255][187], and we plan for the future to validate Severity in industrial setting.

Threats to *external validity* could be caused by the fact that we only analyzed projects written in Java and publicly available. However, we partially mitigate such issues by analyzing 10 projects with more than 22 versions each. Moreover, the high number of CDs could have reduced the effect of the other types of detected AS in the results. We could have mitigated this aspect by sampling the CD instances and thus balancing the dataset. However, this would additionally reduce the size of the dataset, mining the validity of the CD results too. In the future, we aim to extend the study with additional data for the smells and further remediate to this threat.

Finally, concerning threats to the *reliability* of the study, we amply discussed the validation of Arcan results in Section 3.

5.3.5 *Final remarks*

We performed an empirical analysis on 22 versions of 10 projects of two software metrics, Severity and PageRank, in order to evaluate the cost-solving and criticality of AS. We also performed this evaluation with the perspective to better understand if in the ADI computation both the two metrics have to be used or not, if they provide hints on the criticality evaluation of the AS that have to be both taken in consideration.

To conclude, from the analysis of the evolution and correlation of PageRank and Severity we found out that the two metrics tend to be correlated, except for some extreme cases.

It could be useful for developers to analyze the specific cases where AS have high PageRank and low Severity (and vice-versa), since they could indicate smell instances which require a tailored prioritization

rationale: developers may be interested in identifying cases where the smell is easy to solve (low Severity) but in an important part of the system (high PageRank), and choose to refactor this case first; on the contrary, s/he could decide not to refactor a smell difficult to solve (high Severity) and in an unimportant (low PageRank) part of the system. We can assert that such smells are a signal that both PageRank and Severity could be useful to define different refactoring priorities, from different points of view. In particular, PageRank can be used to identify parts of code which need a continuous inspection, while Severity can be used to evaluate the cost-solving for the AS removal.

The smell type presenting the strongest correlation is CD, suggesting that highly critical components (with high PageRank) attract CDs hard to solve (with high Severity). Thus, developers should pay great attention to CD smell, also because CD is the most common AS and in particular those at package level tend to become more critical in terms of PageRank in the history of the project development. However, we do not exclude the possibility that the two metrics have strong correlation because they capture the same aspects of smells. In that case, we could exploit this information to refine the computation of our ADI and leave out one of the two.

In any case, we need to conduct a validation of both metrics and on the correlation results, with expert developers or by comparing the ranking provided by the metrics with information coming from issue trackers [133]. The intuition behind is that a component affected by a critical smell (with high PageRank and high Severity) should be also interested by many issues. Indeed, other studies in the Literature adopt this idea, for instance Le et al. [133] exploited the issues reported in the projects' issue trackers to analyze the impact of smells on software development and the creators of Titan (DV8) use issues as input for the identification of "hotspots", i.e., set of classes affected by many design flaws [116][163].

5.4 SUMMARY OF THE FINDINGS

This chapter offered an overview of the research we conducted in the evaluation of the architectural technical debt of Java projects. In particular we computed an existing index (ADI) over a set of projects addressing different application domains:

- Internet of Things (IoT) platforms
- MultiAgent System (MAS) platforms

We also studied the fundamental component of the index with the support of sensitivity analysis and by analysing the single metrics assessing AS criticality. The three AS that we focused on are Cyclic Dependency, Unstable Dependency and Hub - Like Dependency, i.e., the ones currently included in the ADI computation.

From our analysis, we came to the conclusion that ATD is not only tied to the growth of the size of software projects, on the contrary, there are practices (adopted by developers) that can help in managing the ATD. We found hints of that in software repositories: when manually analysing the points in the evolution of both IoT and MAS platforms where the ADI values reach their minimums, we found evidence that periodical refactorings and frequent *fixes/improvements* to the code are effective in keeping the debt under control. This could mean that architectural debt, usually considered only at architectural level, has also a relationship with issues at code level, such as bugs.

On the other hand, reusing third-party components, which is a recommended practice as much as refactoring in SE, seems to be detrimental for the system's quality. In particular, in the example projects that we analysed, it lead to the accumulation of both AS and TD.

Concerning the different types of smells we detected in the analysed projects, we can say that in general the most present AS in all the different domains is Cyclic Dependency, followed by Unstable Dependency. We found very few examples of Hub-Like Dependency.

We also developed a tool pipeline able to decompose a given technical debt index and indicate to developers the AS which contribute the most to the debt and the most problematic packages.

Finally, concerning the criticality of the considered AS, we found a strong correlation between the values of PageRank, used to evaluate the importance of an AS and Severity, our proxy for AS cost-solving. This means that, concerning the analysed data and the considered smells, the criticality and the cost-solving of smells go hand in hand: if a smell affects an important (unimportant) part of the system, then it will also have a high (low) cost-solving. Future directions concerning this subject can be found at the end of the thesis in Section 8.2.

6

ARCHITECTURAL SMELLS DETECTION IN MICROSERVICES ARCHITECTURES

In the past few years the microservices field has received large attention, both from industrial and academia world [85]. Microservice architecture is an architectural style that structures an application as a collection of small, loosely coupled and self-contained components, called services, which implement specific business capabilities [175]. These components communicate through lightweight protocols and are usually developed by dedicated teams which take care of their entire life cycle, enabling independent deployment. A single component (service) in this architecture is elastic, resilient, composable, minimal, and complete; moreover it is easy to replace it and focused on a single business capability. Services can be developed with different programming languages and by different teams of developers, which makes them ideal in a business environment in continuous evolution. The characteristics of the services enable selective scaling, which means that the number of instances of each service can be chosen and tailored depending on the particular need; moreover they enable continuous and fast delivery. For these reasons, many legacy existing projects are moving from their original monolithic architecture to embrace this new paradigm.

The migration consists in various steps aimed at refactoring and decomposing the current codebase in independent domain components. At the moment, these tasks are usually carried out manually [118] with the partial support of software analysis tools to navigate the code under inspection. In addition to being time consuming, this process requires specialized personnel on software analysis with knowledge about the system to be refactored. Moreover, in large legacy software the documentation of the architecture and code design is often missing or does not reflect the actual implementation. In a survey conducted on 18 practitioners, Di Francesco et al. [65] collected feedbacks on migration to microservices experiences. The questions regarded the activities carried out during the migration: reverse engineering, architecture transformation and forward engineering. Concerning in particular the reverse engineering phase, the majority of the interviewed agreed that understanding the existing system, in particular by identifying its functionalities and subdomains, is very important to architect the new system. Moreover, the authors identified challenges regarding the high level of coupling of the existing system, the problems in identifying the candidate microservices and the system decomposition. They suggest that a tool able to support

practitioners in these activities during migration could be particularly useful.

The final solution (the migrated architecture) could be subjected to smells too. Some of the recent works on migration from monolithic systems to microservices [86][232][147]) highlighted that the migration process generally increases Technical Debt (TD). The major reason behind this is the need to rewrite the vast majority of the code to be migrated, an activity which could expose the system to the introduction of new issues. Moreover, the monitoring of the migration process, the large number of point-to-point connections between services, and the presence of business logic in the communication layer usually increase the dependency between services and consequently the TD [242].

In order to identify which factors can affect TD in microservices-based systems, a set of microservices-specific anti-patterns and smells have been identified [236], [234]. Bogner et al. [37] conducted a Systematic Literature Review on the subject and created a public catalog of anti-patterns/smells. Other practitioners and researchers have also proposed anti-patterns (or smells) [207][7][208], highlighting that they should be removed from the code since they could decrease software maintainability, increase bug-proneness, and generate different types of issues. In this thesis, we call them *Microservices Smells* (MS).

While various tools exist for monolithic systems that can detect code smells and architectural smells, few tools support the identification of microservice smells, which means that developers need to manually check whether their systems comply with standards and do not contain smells. This is due to the usage of recent technologies for the implementation of this kind of architectures and the difficulties that must be faced when monitoring network communications among services at runtime [159].

Our work focus on both the migration and the maintenance of microservices. We conducted two industrial studies about the role of architectural smells during the migration from Java monolithical systems towards microservices and then developed *1)* an extension of Arcan for the detection of MS, still based on the static analysis of the code *2)* A brand new tool, named AROMA (Automatic Recovery of Microservices Architecture), which leverages dynamic analysis techniques to reconstruct the microservices architecture and identify MS.

6.1 INDUSTRIAL CASE STUDIES ON THE MIGRATION TOWARDS MICROSERVICES

Before starting the study of smells in microservices systems, we asked ourselves whether the migration towards microservices was hampered by the presence of AS in the (monolithical) architecture to migrate. Given that the core concept behind the design of microser-

vices architectures is the extreme cohesion and low coupling of the services, not only logically, but also technologically, and given that AS (among the others) hinders those properties, we hypothesized that AS could represent an obstacle for the migration. Hence, in our studies we first analyse the existing codebase to identify AS and then attempt the migration, monitoring the possible difficulties due to the smells. Our approach is summarised in a *process* which we implemented in Arcan (see Section 6.1.1). This extension supports the identification of candidate microservices through different techniques, which vary from graph algorithms to topic detection, where the latter has been previously used in the literature in different contexts such for example to analyze code in the context of public projects/repositories labelling [143, 203, 256]. In particular the application of Latent Dirichlet Allocation [36] algorithm has been used in our approach to identify services depending on the application domain.

Our *migration process* is not the first one proposed. The discussion on how to migrate from monolithic architectures to microservices produced several practical guidelines to help developers in this process: they usually come from direct experiences in the industry [46], but also from research in academia [28] [160] [101]. See Section 7.4.2 for more insights on the related works.

The first study was conducted in collaboration with Alten Italy[1], while the second with Anoki[2], an italian company active in the field of IT consulting.

In the remainings of this section, first we introduce the Arcan extension for the support to the migration, then we report the results of the two studies.

6.1.1 *Candidate Microservice Identification through Arcan*

This Arcan extension offers a set of functionalities to gather information on how to decompose the project starting from the monolithical code. We propose a migration approach through different steps: *1)* architectural smell detection *2)* dependency graph analysis *3)* topic detection (see Figure 6.1). All these steps produce information useful to identify candidate microservices. The three steps differentiate since the first offers hints on how to decompose the project under analysis taking in consideration the presence of architectural smells; the second aims to retrieve blocks of the project that are structurally independent and can be reused or transformed in microservices, while the third aims to identify the parts of the project which belong to the same "domain" in order to return a "semantic map" of the project. In this way a maintainer involved in the migration is able to collect hints and information of different kinds coming from different sources, and

1 https://www.alten.it/, accessed October 2021
2 https://www.anoki.it/, accessed October 2021

choose the decomposition solution which best fit the project under analysis.

In particular, as shown in Figure 6.1, the Dependency Graph Analysis step includes different methods to identify microservices, respectively: connected components detection of the dependency graph ① and generation of two views, Vertical Functionality ② and Logical Layer ③; while the Topic Detection step includes the analysis of the text coming from the code and execution of two *topic detection* algorithm to extract "hidden concerns", named Latent Dirichlet Allocation (LDA) ④ and Seeded Latent Dirichlet Allocation (SLDA) ⑤. At the end of the process, the available information regards the hidden modules in the monolithic architecture that can be exploited for the migration to the future microservice architecture: the proposed solution aims to maximize the modules' cohesion to ease the activity of creating single-purpose services.

DEPENDENCY GRAPH ANALYSIS The aim of this step is to obtain an indication on how the monolithic architecture should be decomposed by looking at the static structure of the project under analysis i.e. its dependency graph. This step is based on an assumption: even if Java monolithic systems are considered a big mixture of lines of code, most of the times they are composed by well defined Java services [28] such as REST services, JMS services, SOAP services, EJB services and Servlet/JSP services. The presence of these services is characterized by the use of dedicated Java libraries which enable their implementation. These can be detected by inspecting the dependency graph with graph queries and by executing graph algorithms. The following paragraphs go deeper in the description of the three methods.

- **Connected Components Detection:** This functionality consists in applying the *Depht First Search (DFS)* algorithm [218] in order to find connected components (sets of Java classes or packages) in the graph by considering the undirected edges. The subgraph that can be generated has only nodes corresponding to the identified components. In Arcan, the algorithm is used to detect totally detached parts of code, which can be extracted independently from the project.

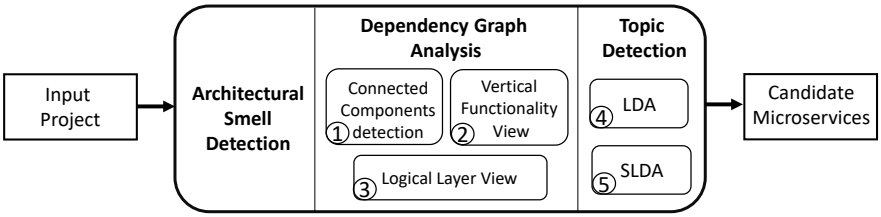


Figure 6.1: Migration to microservices process

- **Vertical Functionality View:** This view aims to isolate and show each functionality contained in the project under analysis in order to support the extraction of the interested parts of code as microservice candidates. This is obtained by running the *Depht First Paths (DFP)* algorithm [218]: by providing a specific set of source classes, the algorithm is able to compute for each source class the paths on the directed graph. The nodes of the paths represent the Java classes and the edges represent the dependencies among the classes. Then, every path is compared one to the other in order to find eventual "shared" classes i.e. classes that belong to more than one path. There are various ways to provide to Arcan the input source classes to be used as starting nodes for functionalities search.

  a) The simplest one, that can be used when the maintainer has zero knowledge about the project under analysis, is to choose the classes with no incoming dependencies. This means that such classes are never referred from other parts of code in the project, making them candidate entrypoints.

  b) The second way requires more information: it chooses classes with no incoming dependencies which refer to specific libraries. For instance if a class exploits the Java API for RESTful Web Services(JAX RS)[3], it may be a good candidate to find an hidden REST service inside the monolithic architecture. The tool already recognizes the libraries which implements the JEE Specification.
- **Logical Layer View:** This view allows to divide the classes in groups depending on the layer they belong to. *Layers* refers to the ones of the three tier model, which organizes the code in presentation layer, application processing (business) layer, and data management (persistence) layer. The tool is able to separate and assign each class to its layer by looking at their external dependencies, in particular checking the Java implementation packages of the JEE specification. Unlike the vertical functionality view, the layered one offers a coarse grained representation of the project under analysis. In this way it is possible to understand the role of each class when the maintainer has no information about how the code is organized.

TOPIC DETECTION Usually microservices are created depending on specific "domains" or "business concerns" of the project. When migrating from a monolithic architecture, it is not trivial to automatically extract such concerns from the code without human supervision. However, a possible solution to this problem could be reached through topic detection techniques, by considering code as text and by looking for *topics* that could correspond to services. In this work,

3 The Java API specification that supports the development of RESTful web service

the algorithms exploited to extract topics from code are *Latent Dirichlet Allocation (LDA)* [36] and *Seeded Latent Dirichlet Allocation (SLDA)*, a semi-supervised variant of the original LDA algorithm [112]. The latter algorithm allows the maintainer to provide some *seed* words so that the model is encouraged in finding evidence of some "expected" topics in the data. The idea behind the choice of the seeded algorithm is that developers may know some of the topics which could be hidden inside the monolithic system and enhance the results of the detection. The following paragraph describes the topic detection process.

1. **Document collection**: a document is created by selecting comments and source code words from a single Java class, in particular the class name, its membership package name and the name of all its methods. Class attributes and variables are not included since often they do not distinguish a class from each other by belonging to a specific topic (e.g. "filename", "x", "a", "temp"). This step is implemented in Java language.

2. **Preprocessing**: this step consists in manipulating the text contained in the documents to enhance the results of topic detection. In particular, the documents created starting from Java classes are *tokenized* i.e. their stream of characters is broken into words. After tokenization, *filtering* is applied. The resulting tokens are converted to lower cases and are analyzed in order to remove numbers, punctuation and stop words, which are the very common words in a language. This step is implemented in Python language.

3. **(Seeded) Latent Dirichlet Allocation**: the last phase is the running of the topic detection algorithm. In order to run the LDA algorithms, the Python library *guidedLDA*[4] was used. This library was chosen because it lets the maintainer to define a set of *seed topics*. The output consists in the detected topics represented as word-topic distribution and the document-topic distribution, that is the proportion of words of each topic associated to a given document.

At the end of the process, the maintainer can collect hints about the semantics of the project to be migrated, in particular on which Java classes are associated to a specific domain.

Notice that this approach differentiate from the *feature graph* introduced in Section 2.1. Even if both leverage Natural Language Processing (NLP) techniques, the model described in this section requires a set of input parameters in order to run, one of which is the number of expected topics (candidate microservices) to extract. The Arcan extension into which the model is implemented is prior to the development

4 https://guidedlda.readthedocs.io/en/latest/, accessed October 2021

of the feature graph and represents our first attempt to exploit NLP in combination with classic software analysis. The model performed satisfactorily in the context of our industrial studies since we had knowledge of the projects under analysis (both studies involved developers actively working on the projects) and thus we had enough information to predict the number of expected topics. However, we strongly praise the benefit of non-parametric procedures, as the computation of tf-idf exploited in the generation of the feature graph, which frees us from the definition of the input parameters and support the software comprehension even when there is lack of project knowledge.

6.1.2 *1st Case study: Alten Italy*

We started a collaboration between academy and a company (Alten Italy) in order to experiment how Arcan could be useful in order to support the migration towards microservices of a project of the company[5].

The outcome of this study was useful both for academic and industrial purposes, since the feedback on the tool were exploited to improve the tool and for the company to identify a useful support to be used during the migration process.

The analysis was carried out by an experienced developer which executed Arcan on an industrial project and identified candidate microservices basing on the tool outcomes. Moreover, he provided several feedbacks on the migration techniques offered by Arcan and on the final candidate microservices solution that he was able to define thanks to the tool. The industrial project analyzed is a Java enterprise project developed to manage the collection of information for the initiation of legal proceedings. It is composed by 267 classes divided into 27 packages. The developer originally took part in the development of the analyzed project, in particular he managed the collection of requirements and the development process. Hence, he possessed remarkable knowledge on the design choices and business logic: we chose this particular case study because we were interested in valuable feedbacks on the quality of the solution proposed by Arcan.

The following paragraphs show the results obtained through the different migration steps implemented in Arcan. The developer followed the approach described in Section 6.1.1. The data generated by the tool are available[6].

5 A publication was extracted from this study [198], in collaboration with Andrea Maggioni

6 https://drive.google.com/drive/folders/1kLJXMPXhG2U8pIrqG_MWVU0STSCdIsMN?usp=sharing

Table 6.1: Detected Architectural Smells

# Unstable Dependency (UD)	# Hub-Like Dependency (HL)	# Cyclic Dependency (CD)		# Feature Concentration (FC)
		class	package	
10	1	4	2	22

ARCHITECTURAL SMELLS DETECTION RESULTS First, the developer executed Arcan. Table 6.1 shows the number of AS detected in the project under analysis. The considered AS were Unstable Dependency (UD), Hub-Like Dependency (HL), Cyclic Dependency (CD) and Feature Concentration (FC) (see Section 2.2). He could retrieve the most relevant information from the analysis of Cyclic Dependency and Feature Concentration smells, for the reasons described below.

Cyclic Dependency The developer recognized four cycles as real issues for the monolithic application. However, he reported that *"Those cycles will not be a problem during the migration"* except for one cycle on classes. This particular smell involved 3 classes which are part of the central logic of the application, whose aim is to create entries on a calendar basing on a set of deadline rules. He foresaw that in the new architecture this logic will be completely redefined, in particular it will be divided into different services. He indicated the presence of the cycle as a possible obstacle to the decomposition of the application.

On the other hand, two of the four CD smells detected on classes resulted to be false positives. Both are cycles between an anonymous class and its corresponding container class and this Java feature always leads to the introduction of a tiny cycle.

Feature Concentration The developer found the detection of this smell particularly useful. He was able to identify the main domain *entities* of the application, that represent the information managed by the application, since the smell instances affected the packages containing business application classes. Table 6.2 shows the identified entities. The approach he followed to identify entities starting from FC smell consists in: *1)* spotting the affected packages from Arcan results; *2)* exploiting the Neo4J browser to navigate the disconnected subgraphs and *3)* extracting the entities associated to the different subgraphs.

DEPENDENCY GRAPH ANALYSIS RESULTS The following paragraphs show the results of the service detection using the *Vertical Functionality* and the *Logical Layer* views. The results of Connected Components detection are not discussed since the developer did not use it to build the final microservices solution; the detected components did not gave him interesting hints on the business concerns/functionalities.

- **Vertical functionality view results** The developer chose to run the generation of the vertical view with the two possible kinds of input offered by Arcan: classes with no incoming dependencies and classes with no incoming dependencies depending from specific JEE libraries. The view generation with the first type of input returns a csv file containing all the directed paths starting from the classes without incoming relations. In this case study the total number of detected paths was 69: the developer found the use of this information expensive in terms of time, hence moved forward with the next analysis. The second type of input computes DFS paths from nodes which have been identified as *Web* and *Web Service* in JEE Specifications. The results of the second view generation returned a total of 3 paths. In this case, he reported that one of the paths was useful during the analysis; it helped in identifying the service regarding the components which manage the entity called *Attachments*, where *Attachments* represents the files uploaded on the application and saved on a Mongo Database.
- **Logical layer view results:** Table 6.3 shows the results of the service detection process using the *Logical Layer View* functionality. The table shows the different layers and the number of classes assigned to each layer and the value of True Positives (TP) and False Positive (FP) class assignments, which the developer used to compute Precision.

  By analyzing the *false positive* results, the developer reported that Arcan can not assign the correct layer to the classes which use the Spring framework [229] classes, both for the Persistence layer and for the Web layer. The matching rules implemented in this first version of the *Logic Layer Detection* algorithm have been thought basing on *old* functionalities of the Java Enterpise Edition, which are used in many legacy projects. In more recent Java application Spring is a popular framework, hence the developer suggested us to introduce new rules taking in consideration the use of Spring to achieve higher *precision* value on the layer-class assignation.

TOPIC DETECTION RESULTS This step consisted into two main parts. First the developer ran the *Document collection* generation of Arcan, which reads the Java source files and produces for each class a csv file which contains the meaningful words contained in the class. In total Arcan produced 267 files which contain 418 different words. Then the developer executed the two versions of the LDA algorithms.

From the first run of the classic LDA algorithm, he noticed the noise produced by some words belonging to technical aspects of the libraries used in the application e.g. the HTTP methods connected to the "Spring Controller" of Spring Framework. Hence, he excluded

Table 6.2: Main Entities

Entity	
Event	User
DeadlineItem	Suspension
Attachment	Notification
Society	Proceedings
CronologyChange	

Table 6.3: Logical Layer Results

Layer	Number of Classes	EVAL	
		TP	FP
Persistence	1	0	1
Web	8	6	2
Core	267	207	60
Precision: 77,2%			

119 words from the vocabulary and added them to the stopwords file. This because he was interested in retrieving information referring to the business logic contained in the project respect to the technical one. The excluded words can be consulted in the available folder[7]. Once the developer modified the stopwords file, he proceeded with the run of both LDA algorithms and compared their results. Both algorithms needed a parameter setting as input, in addition to the document collection. The choice of all the parameters except for the number of topics (which was chosen by the developer) was guided by the state of the art of the topic detection field [98]. The parameters are:

- **number of topics**: 10
- **alpha** - prior weight of each topic in a document: 0.01
- **beta** - prior weight of each word in a topic: 0.1

Latent Dirichlet Allocation results: Table 6.4 contains the 10 topics retrieved by the classic LDA algorithm. The developer found the results interesting since the detected topics contain many words that recall the entities and functionalities of the application. For instance he found references to the functionality of the application which sends an alert when a *proceeding* is created by reading the words of topic 8: "creation", "proceedings", alert". Another example, words of topic

7 https://drive.google.com/drive/folders/1kLJXMPXhG2U8pIrqG_MWVU0STSCdIsMN?usp=sharing

4 "user", "change", "roles" recall the application feature of changing the roles of a *user* inside the application.

Moreover, he was able to obtain the same information on the entities identified with the AS analysis (Table 6.2): he could label each topic (see column Entity in Table 6.4) and became aware of a new entity which, basing on his past knowledge, he called *Key-Value*.

Seeded Latent Dirichlet Allocation results: in order to run the modified version of LDA, the developer defined 5 *Seed Topics*. He defined 4 of them on the basis of the entities collected through the AS detection step, while one (*Summary)* represented an entity expected by the developer:

1. proceedings, deadline, suspension, item (**Deadline**)
2. summary, voice, comment, attachment (**Summary**)
3. society, ragione, sociale, soggetto (**Societies**)
4. event, reminder, days (**Events**)
5. notification, recipient, sender, object (**Notification**)

Table 6.4 contains the results of the Seeded LDA analysis. The execution of SLDA was not considered useful by the developer because even if the algorithm retrieved quite the same information from the execution of classic LDA, he could not easily label each topic with a corresponding entity. Moreover, the *Summary* entity was not identified as expected by the developer (see Section 5.3.3).

After comparing the results coming from all the methods implemented in Arcan, the developer produced the final solution. Table 6.5 shows the candidates microservices, for each service there is a brief description of the functionality associated to it. As a results of the topic detection step, the developer chose to incorporate entity *Event*, *Deadline* and *Suspension* into a unique candidate microservice. Moreover, he introduced *Key-Value* and discarded *CronologyChange* on the base of his past knowledge on the project.

DISCUSSION We now discuss the results and feedbacks obtained from this case study on the microservice migration process through Arcan. The developer ran Arcan following the steps described in Section 6.1.1 in order to identify how many "business services" compose the industrial application under analysis. The AS detection was the preferred and most useful step for the developer in order to understand how the application was composed. He was able to identify the parts of code related to single *entities* which could become microservices (Table 6.2). Moreover the AS detection made him aware of a problem regarding a specific entity named *Deadline*: the creation of a *Deadline* requires the information present in *Suspension* and *Proceedings* and vice-versa, part of the problem was solved by incorporating entity *Deadline* with *Suspension*, while the Cyclic Dependency

Table 6.4: Topic Detection results

	LDA		*Seeded LDA*
	Topic	**Entity**	**Topic**
1	proceedings deadline suspension attachment start state days management	Proceedings	1 proceedings deadline date suspension reminder item payment days
2	user history username process provvedimento email finale change	User	2 summary date proceedings comment voice data event description
3	delibera subject collegio approvazione audizione provvedimenti area action	Key-Value	3 history event activity process analize society ragione date
4	user change roles summary user change roles summary	User	4 state proceedings attachment society management document visibile event
5	access data impegni decisoria payment avvio procedimento turnover	Key-Value	5 attachment cronology history object change interceptor resolver changed
6	attachment proceedings reminder today date events notifications recipient	Notification	6 subject impegni decisoria provvedimenti action istruttoria procedimento turnover
7	proceedings deadline event summary date voice comment item	Suspension	7 user finale roles username email provvedimento data role
8	event state proceedings history creation proceedigs alert assigned	DeadlineItem	8 delibera collegio access approvazione documents audizione data ammissibilita
9	documents audizione data ammissibilita appeal atto determina document	Key-Value	9 event deadline proceedings voice summary events simplified owner
10	society data activity ammissibilita files status ragione sociale	Society	10 user date change today event audit state expire

Table 6.5: Candidates Microservices

Candidates Microservices	
Proceedings	This service will manage the Proceedings, the main entity of the new system.
Attachment	This service will manage the Attachment, an Attachment is a file associated to a Proceedings.
Society	This service will manage the Societies which could be associated to a Proceedings.
User	This service will manage the User authentication and the applications roles associated to a User.
Notification	This service will manage a chat service.
Deadline & Suspension	This service will manage the *Deadline & Suspension* logic.
Key-Value	This service will manage a new type of entity called *Key-Value*; this entity will have only a few attributes(e.g. id, value, type). A *Key-Value* will be used by the Front-End part to display show some select tag at the final users.

between *Deadline* and *Suspension* should be analyzed and possibly removed during the migration process in order to decouple the services. The Dependency Graph Analysis is the step which gave him less information, because the implemented methods are based on the idea that the application under analysis refers to a JavaEE standard architecture used in many legacy projects. The analyzed application is based on *SpringFramework* [229], so Arcan could not assign the correct layer to the classes and put all of them in the *Core Layer*. Moreover he did not use the results coming from the Connected Component detection, because the microservices candidates proposed by the algorithm were not in accordance with his background knowledge. This tells us that in general we have to improve our current approach about graph analysis. Finally, the developer validated the topic detection step. He preferred the classic version of the LDA algorithm since in his opinion the resulting topics were more relevant respect to the seeded version. He supposed that the seeded LDA results are strictly connected to the chosen seed topics. The topic detection confirmed the results of the AS detection and provided additional information useful to establish the final solution (Table 6.5).

In conclusion, the developer stated that *"In general the migration process is not easy to carry out, since a deep knowledge of the project subjected to the migration is needed in order to have significant results. Arcan can be very useful: to retrieve knowledge about the project using the architectural smell detection and the vertical functionality view, and to extract more information about the services using the LDA algorithm."*

FINAL REMARKS AND LESSONS LEARNED We collected important lessons learned from the collaboration with the industry. First of all, (1) we received positive feedback concerning the usefulness of the Arcan tool, which stimulate us to continue working in this direc-

tion and increase the collaboration with industry in this context; (2) we collected several useful feedback to enhance and extend Arcan; (3) we understood that the analysis of some data are more time consuming than other, such as the information provided by the architectural smells detection and the dependency graph analysis with respect to topic detection. (4) We observed that AS detection and dependency graph analysis are suitable for a deep project comprehension, while topic detection could be exploited for the initial understanding of the project, when few knowledge is available to the practitioners. However, in the case study presented in this paper, topic detection results enhanced when the developer changed a setting (*stopwords* file) and executed the algorithm again: this suggests that the topic detection functionality works better when applied across multiple iterations. All these findings could lead to the refinement of the current migration approach to fully exploit the potential of Arcan functionalities. Moreover, the current approach addresses only a step of the migration to microservices i.e. the information extraction from the current system. We aim to extend our work in order to support the concrete implementation of the services and provide a method to evaluate the quality of the migrated architecture, as studied by Carrasco [52]. Having a framework to evaluate the software quality before and after the migration could assist in making decisions during the migration phase.

6.1.3 *2nd Case study: Anoki*

The second case study was conducted in collaboration with Anoki[8], an Italian software consultant company which develops mobile and web applications specialized in open banking and educational platforms.

In particular, the analyzed project was a Business Management System with a monolithic architecture, written in Java. The project was 10 years old and can be considered a medium-large project with 1343 classes and 112 packages.

The team working on it at the time of our study was formed by three developers: the first one was a junior developer with 4 years of experience during which he has been working on the project at hand, the second one was a middle developer with 9 and a half years of experience of which 1 year and a half spent working on the project in object, while the third one, the team leader, was a senior developer with almost 15 years of experience that has been working on the project for 2 years. In the circumstances of this study we also validated Arcan results (see Chapter 3).

8 This work was part of the master thesis of Federico Locatelli, who we thank for the hard and successful work he did.

We analysed with Arcan six versions of the project that the developers indicated as particularly problematic or linked to refactoring sessions. Along with such versions, we analysed the latest version of the project at that moment.

In Table 6.6 the total number of detected instances for each architectural smell are indicated.

Table 6.6: Anoki analysed versions

V	CD-Cl	#CD-Pkg	#HL-Cl	#HL-Pkg	#UD	#FC	#SF	#GC
10.0.1.1	135	5	3	3	19	4	81	10
9.0.5.0	107	7	3	4	20	8	79	10
9.0.4.0	106	9	3	4	20	7	77	10
8.2.1.0	83	9	3	3	19	4	67	10
8.2.9.9a	70	9	3	3	19	4	71	10
8.1.0.0	48	7	3	3	19	3	74	11
6.5.2.0	6	3	2	1	12	1	48	4

Acronyms. V: version, Cl: classes, Pkg: packages, CD: Cyclic Dependency, HL: Hub-Like Dependency, UD: Unstable Dependency, FC: Feature Concentration, SF: Scattered Functionality, GC: God Component

MICROSERVICES IDENTIFICATION We now describe the identification process for each microservice. In particular we describe *1)* the Feature Graph Analysis, *2)* the Connected Components analysis *3)* the Vertical View analysis *4)* Dependency Graph analysis *5)* the Topic Detection identification.

At the end of the analysis description, *for each microservice*, we propose the division of the analyzed classes into three categories: *1)* the classes belonging only to the described microservice, *2)* the classes that should be duplicated (completely or partially) across more than one microservice and *3)* the classes belonging to other microservices that used or were used by classes of the described microservice. We made this division to have a clear vision of the microservice bounds and give to the developers as much information as possible about it.

With respect to the first case study (see Section 6.1.2) we added the Feature Graph analysis, i.e., we analyzed the Feature Graph (see Section 2.1) produced by Arcan containing each feature of the project represented as a node in order to check if there was any feature node with a name that could recall the microservice we were looking for. Moreover, before consulting the Arcan results, we asked to the developers what microservices they expected to extract from the existing codebase. That is why we present the results separately for each

microservice. Finally, we monitored the time spent to identify each microservice.

Report Engine microservice

The first microservice that we identified takes care of all the reporting features of the system and that was indicated by the developers as one of the easiest to identify in their opinion. Identifying this microservice took about *8 and a half hours*.

1. the **Feature Graph Analysis** took *about 20 minutes*. We queried the graph using the words *"report"* and *"engine"* and their stem (root) forms, finding three nodes called *"engin"*, *"report"* and *"repo"*: these feature nodes contained an overall of 7 classes of which 6 had dependencies among each other. Instead, the remaining one was completely isolated. All of them appeared to belong to this microservice at first sight.

2. the **Connected Components consulting** took *about 20 minutes* and showed that 2 classes formed a connected component while the other ones were part of the largest connected component, that we defined *monolithic connected component* (i.e. the connected component containing most of the project's classes in the monolithic architecture). Interestingly, one of the two isolated classes is the implementation of one of the classes included in the large connected component, meaning that, although there is not a real dependency between them, they are correlated and they should be part of the same microservice. This is an important consideration because it suggests that the relationship between two classes should be evaluated not only by looking at their dependencies, but also looking at the type of class and their responsibility (e.g. one being an interface and one being its implementation).

3. the **Vertical View analysis** took *about 20 minutes* and confirmed our ideas about the already identified classes, additionally giving us a wider view of the part of project we were examining. We also noticed some classes and packages with names recalling the microservice's concerns inside the vertical dependency path and we considered them worth analyzing.

4. the **Dependency Graph analysis** took *about 3 and a half hours* and helped us to establish the microservice bounds (the set of classes actually belonging to it). In fact, it allowed us to analyse the dependencies of the already identified classes and classify the classes based on our opinion of their relationship with the microservice. During this step, we also noticed that most of the classes we considered part of the microservice belonged

to sub-packages of the same package. Focusing on that package we discovered that all the classes (except one) contained in it or in its sub-packages were exclusively part of this microservice and they were also the only ones with such characteristic. Thus, in this case the identification was easier, because the classes belonging to the microservice were already all part of a unique package whose name was also recalling the microservice name. This proves that a package-by-feature structure really enables a migration to microservices process. During this phase we also spotted classes that were put in a package accidentally and should have been moved into other packages.

5. the **Topic Detection** took *about 3 hours* and in this step we recognized one topic as the more tied to the microservice, since it contained all the 3 tokens recalling its concerns: *"report"*, *"reports"* and *"engine"*. Moving forward to examining the topic distribution file, we noticed that the **Topic Detection granularity was too coarse-grained** as the topic contained a high number of tokens, meaning it was correlated to very different domains and classes and, thus, making this kind of analysis useless in a microservice identification process. So, we tried to modify the parameters of the topic detection algorithm by increasing the number of topics that had to be detected from a range between 2 and 11 to a range between 21 and 22, because these were the ones that gave us the best result for the microservice (i.e. there was a topic containing, with the highest frequency, all the tokens we were interested in). Thus, by re-calibrating the detection we were able to check our evaluations with a positive outcome, as the classes we considered part of the microservice had the highest probability of being part of the topic in question (compared with the other classes). Adjusting the detection took us most of the time spent on this step, so it is a very time-consuming activity.

In brief, the classes being part only of this microservice are 66, the classes that should be duplicated are 12 and the classes belonging to other microservices that used or were used by classes of this microservice are 10. The developers gave a positive feedback on the result of the identification of this microservice.

Scheduler microservice

The second identified microservice performs batch operations (i.e. operations that can be run in background) when invoked. It was indicated by the developers as one of the easiest to identify. Identifying this microservice took about *4 hours*.

1. the **Feature Graph Analysis** took *about 15 minutes* and demonstrated to be valuable starting point once again, even though we

could have obtained the same result by looking for the name of the microservice among the package names. In fact, querying the graph using the word *"scheduler"* and its stem (root) form we found two nodes, both called *"schedul"*. The identified nodes contained one class each, with one being the implementation of the other and both belonging to sub-packages of the same package. Such package captured our attention because its name was the same as the microservice's (*"scheduler"*). Both the feature nodes also had a dependency with another node called "proxy" that contained two classes belonging as well to sub-packages of the "scheduler" package.

2. the **Connected Components consulting**, which took *about 10 minutes*, showed that all the already identified classes were part of the monolithic connected component. In the case of this microservice, this step only suggested us that the microservice was not already isolated from the rest of the architecture and its bounds needed to be manually identified.

3. the **Vertical View analysis** took *about 5 minutes* and unfortunately it was not useful: only one path, starting from an already identified class, was detected by Arcan and that path contained only other already discovered classes.

4. the **Dependency Graph analysis** took *about 2 hours* and during this step we managed to determine the microservice bounds. However, after a discussion with the developers, we realized we missed some classes. The project leader reported a group of classes that should belong to this microservice too, since their concerns are the same as the microservice's. Such classes were not found during the previous analysis as they were completely isolated from the rest of the microservice.

5. the **Topic Detection** took *about 15 minutes* and in this step we recognized two topics containing the token *"scheduler"*, both with a low term frequency. This poor result was caused by the fact that this microservice is much smaller than the *"Report Engine"* microservice, the service on which the topic detection was calibrated. The granularity problem still persists in this case, suggesting that the number of expected topics should be adjusted for each microservice, unless they are all of the same size in terms of classes/packages. However, re-calibrating and re-running the topic detection algorithm would cause a great loss of time. For this reason, we decided to keep the same detection granularity for all the successive microservices. In this case the coarser detection granularity made it impossible to check our considerations on the classes belonging to the microservice.

In brief, the classes belonging only to the `Scheduler` microservice are 7, the classes that should be duplicated are 11 and the classes belonging to other microservices that used or were used by classes of this microservice are 8. The developers gave a positive feedback on the outcome of our identification of this microservice.

File Manager microservice

The third microservice to be identified is responsible, as the name implies, of the project's file management. It was indicated by the developers as one of the easiest to identify in their opinion. Identifying this microservice took about *7 hours*.

1. The **Feature Graph Analysis** took *about 10 minutes* as we queried the graph using the words *"file"* and *"manager"* and their stem (root) form and discovered two feature nodes named *"manag"*. They were associated to 2 classes with no dependencies between each other. Even if this outcome was small, it was enough to obtain a starting point for the identification.

2. During the **Connected Components consulting**, searching the classes found at the precedent step inside the connected components detected by Arcan took *about 15 minutes*, revealing that the two classes were both part of the large, monolithic connected component. While exploring the different connected components we noticed many classes whose names recalled the microservice domain, all belonging to the monolithic component, so we considered them worth to be analyzed in the following steps.

3. the **Vertical View analysis** took *about 20 minutes* and gave us a clearer idea on how the already discovered classes were related by examining the 2 vertical paths detected by Arcan that start from those classes.

4. the **Dependency Graph analysis** took *about 1 and a half hours* and allowed me to establish the microservice bounds without particular difficulties.

5. the **Topic Detection** took *about 1 hour*. First of all, we decided to enhance the topic detection algorithm by adding 66 new words to the list of the detection stopwords. The new words represented Java and frameworks key-words and other terms that had nothing to do with features of the system, so **ignoring them made the Topic Detection results easier to comprehend and more useful for the identification process**. In fact, during the Topic Detection step of the previous microservices, we noticed that many tokens were insignificant for our goal. we also removed the word *"file"* from the stopwords as it was useful for the specific case of this microservice. After these changes,

> we found one interesting topic containing the tokens *"file"* and *"management"*, even though they were not the ones appearing with the highest frequency because of the detection granularity not being adjusted for this microservice (which is smaller than *"Report Engine"*, the service on which the calibration was tailored). For the same problem, looking into the topic distribution file was not useful to confirm or deny our evaluation, but at least we were able to corroborate that the classes we considered part of this microservice were present with a high probability inside the topic in object.

Before elaborating the final result, we discussed the gathered informations with the developers. They pointed out that a group of 3 classes we did not consider were related to the microservice in the user interface, inside the JSP (JavaServer Pages) pages and Javascript files, and in the SQL database. A JavaServer Pages page is a text document which constructs dynamic content [60]. The mentioned classes were not taken into consideration as they did not have dependencies with other classes belonging to the microservice and because Arcan does not include the GUI and database analysis. This issue could be resolved by adding to Arcan the possibility to analyze JSP, Javascript files and the SQL database and let the user looking for relationships between them and the Java classes.

In brief, the classes belonging only to the microservice are 23, the classes that should be duplicated are 18 and the classes belonging to other microservices that used or were used by classes of this microservice are 3. The developers gave a positive feedback on the outcome of the identification of this microservice.

FINAL REMARKS AND LESSONS LEARNED We now resume and discuss the lessons learned from the experience we had in extracting the microservices from the Anoki project.

In general, we acknowledged that the process and features proposed by Arcan are useful, but with some limitations.

Concerning the Topic Detection feature, we reported in the description of the `Report Engine` extraction the *granularity issue*, i.e., the topic that we found more related to this microservice was too coarse-grained. In particular, the topic was correlated to very different domains and classes and the information it provided was useless for the microservice identification. This was due to the settings of the detection algorithm, programmed to identify few topics and to associate many tokens to them. After re-calibrating the detection algorithm for the `Report Engine` microservice, the issue was fixed for this microservice, but persisted for the successive microservices, probably because they had different sizes (more classes/packages). Taking into account the fact that adjusting the detection required a great amount of time and considering the low relevance of the Topic Detection results in the

identification process, we concluded that for future case studies the best solution is to calibrate the detection algorithm in order to detect a large number of topics (but with more precise class/tokens association) before starting the whole microservices identification process. In this way, all the microservices would be associated to topics coming from the same model and the Topic Detection could be run only once, without re-running it for each microservice. This at the expense of a less precise microservice identification as every microservice would have a fixed number of associated tokens, not reflecting the actual feature dimension.

Turning to another lesson learned, the `Project` microservice identification, on the other hand, showed that the right communication is fundamental while working on a microservices extraction process: often, the disagreement between the developers and us about this microservice was only due to misunderstanding. Indeed, the developers gave a very positive feedback on the results of the microservices identification implying that identifying microservices using Arcan makes the process easier and allows to obtain good results.

Thanks to the examination of the metrics extracted from the results of the microservice identification (i.e. Time needed for the identification of each microservice and the microservices size), we could make another important consideration: the experience with the identification process and with the usage of Arcan, as much as the microservice size, affect the time required for the identification. In fact, `Report Engine` is the microservice which required the longest time to be identified whilst being the third largest one, probably because it was the first one to be identified. Anyway, it is right to mention also the fact that, during its identification, Arcan's Topic Detection was adjusted, causing the loss of a lot of time. `Scheduler` microservice, on the contrary, was the smallest of all the identified microservices and, as expected, required a very short time compared to the others, while `Notification` and `Project` took respectively the second and the fourth longest time to be identified despite being the largest ones, probably because they were also the last ones to be identified and we were more experienced at that time.

Finally, the actual application of Arcan in the identification process showed that analyzing the feature graph is a valuable starting point and that the dependency graph analysis is the core step of the process. In particular, the required (manual) effort to examine the classes dependencies is significantly reduced compared with a traditional dependency analysis, thanks to the graph opening to consultation and querying. The other steps and the corresponding Arcan functionalities were less relevant in the process, but they have proven to be helpful to double-check the results of the dependency analysis.

6.2 TOWARDS MICROSERVICE SMELLS DETECTION

We now introduce the two strategies, based on static and dynamic analysis respectively, for microservices smells detection. The first one is implemented in Arcan, extended specifically for this purpose[9]. It allows to statically analyse projects implemented with few, specific technologies and frameworks. In particular, the tool is able to analyze only projects in Java programming language and to retrieve dependencies only declared with the Spring [229] *RestTemplate* interface. This is why we developed also AROMA (Automatic Recovery of Microservices Architecture), able to reconstruct the MS architecture as a directed graph without being bounded to a specific programming language. It exploits dynamic analysis: in particular it relies on the Zipkin distributed tracing system [10] to dynamically collect information about API calls, service names and network attributes. On top of the recollected architecture, we can detect a set of MS smells.

Both strategies are at an early development stage and we had few opportunities to validate their results, mainly because it is very difficult to find both Open Source and industrial microservices projects suitable for analysis. However, in the sections dedicated to each tool we report the results on few toy examples which exemplify the tools' workings.

6.2.1 *Microservice Smells identification - Arcan extension*

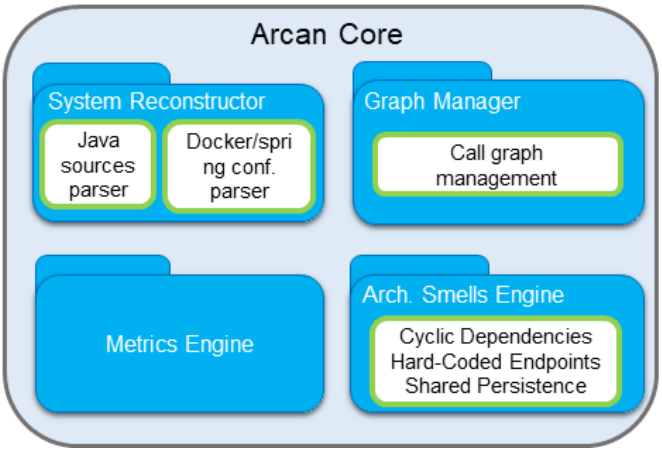
In this Section, we describe the detection strategies of three microservices smells: *Shared Persistence*, *Hard-Coded Endpoints*, and *Cyclic Dependency*.

As described in [19], Arcan consists of four components, which manage the different steps of the provided analysis: Figure 6.2 shows these components with the new additions. To extend this tool and make it suitable for the detection of microservice smells, we *1)* added new parsers to the component dedicated to architecture reconstruction (*System Reconstruction*) in order to scan Java source files and docker/Spring configuration files. Moreover *2)*, we developed three new detectors, one for each microservice smell, and added them to the component that collects all the architectural smell detectors (*Architectural Smells Engine*). As described in Section 2.1, Arcan relies on graph database technology: All the computations are based on the *dependency graph*, which is the representation of the project under analysis in the form of a directed graph. Currently, the tool allows storing the graph in a Neo4j [173] graph database, which also offers a browser for visualizing and querying the graph. We extended the dependency

9 A publication was extracted from this work [197], in collaboration with Valentina Lenarduzzi and Davide Taibi

10 Zipkin is a distributed tracing system. https://zipkin.io/

Figure 6.2: New Arcan core components for the detection of microservice smells



graph representation (*Graph Manager*) in order to include microservices and called it *call graph*: each node represents a microservice and each edge represents a microservice call.

Some of the proposed detection strategies are limited to specific programming languages, technologies, and frameworks. We chose a selection of them depending on two constraints: availability of compliant open-source projects and ease of automation of the detection strategy. Next, we will describe each strategy by providing its *definition* and the description of the related *detection*.

CYCLIC DEPENDENCY *Definition:* A cyclic chain of calls among microservices. e.g., A calls B, B calls C, and C calls back A. Microservices involved in a cyclic dependency can be hard to maintain or reuse in isolation.

Detection: Arcan already automatically detects Cyclic Dependencies in monolithic Java applications by exploiting the dependency graph representation and graph algorithms. For the detection in microservices, we exploit the newly introduced *call graph*. Hence, in order to reuse the original Arcan detector, we implemented the identification of microservices dependencies. Microservices communication can be managed in different ways depending on the technologies, frameworks, and strategies that have been used; hence the call graph generation must be adapted depending on each specific case. In this paper, we focus our attention on Java projects exploiting the Spring framework [229] and/or the Docker platform. The nodes representing the different microservices are generated automatically from the information provided by the configuration files. In particular, the considered Docker files are `docker-compose.yml` and `Dockerfile`; the used Spring file is `application.yml`: they are all useful for service names identification. Concerning the microservices dependencies identifica-

tion, in Spring applications a class named *RestTemplate* is used to handle http requests from a service to the others, hence we detect its usages to identify services communication. For the same reason, Arcan looks for the usage of *Feign* [74], which is a Java library for web services development compatible with Spring.

Dependency detection:

- **Input**: Java project folder
- **Exec**: Scan {docker-compose.yml, Dockerfile, application.yml} files to find microservices names
  - *FOR EACH* service_X, explore service source files and look for pattern matching of Feign annotations and *RestTemplate*.
    - *IF MATCH* dependency_pattern = "@FeignClient(name = "service_y")"
      - dependency = $service\_x \rightarrow service\_y$
    - *IF MATCH* dependency_pattern = "RestTemplate"
      - get service_y name from the methods of RestTemplate class.
      - dependency = $service\_x \rightarrow service\_y$
- **Output**: A file with the detected dependencies

Cyclic Dependency detection: Once the microservice dependencies are identified, the output results of the algorithm are processed by Arcan, which creates the dependency graph. Nodes represent microservices and edges represent their dependencies. Thanks to the graph representation, it is possible to detect Cyclic Dependency smells.

- For each microservice, create a node and add it to the call graph.
- For each microservice call, create an edge to model the dependency.
- Run the Depth First Search (DFS) algorithm: each detected cycle is an instance of a Cyclic Dependency smell.

HARD-CODED ENDPOINTS *Definition:* Hard-coded IP addresses and ports of the services between connected microservices. This smell leads to problems when the service locations need to be changed [236]. *Detection:*

- Scan the source code and look for pattern matching. The first part of the pattern (until the semicolumn) identifies IPv4 addresses and the second part matches ports.
  - pattern = \b\d1,3\.\d1,3\.\d1,3\.\d1,3:(6553[0-5]|655[0-2][0-9]\d|65[0-4] (\d)2|6[0-4](\d)3|[1-5](\d)4|[1-9](\d)0,3)\b

This strategy can be applied to every microservice implementation since it does not depend on any specific technology or framework. In this study, we experimented with the detection in Java projects.

SHARED PERSISTENCE *Definition:* Different microservices access the same database. In the worst case, different services access the same entities of the same database [236]. This smell highly couples the microservices connected to the same data, reducing team and service independence.
Detection:

- For each microservice, collect all database references/usages (database name or database url).
- If two or more different microservices refer to the same database, then those services are affected by the smell.

The detection of this smell is strictly linked to the technologies used. At the moment, we are focusing our attention on Java projects exploiting the Spring framework [229], which allows storing configuration information regarding databases in dedicated files (*YAML*[11] and *.properties* files) and we only detect accesses to the same databases (not entities) i.e. we look for the same connection string.

6.2.2 *Validation - Arcan extension*

In this Section, we report on the validation of the proposed detection strategies on open-source projects.

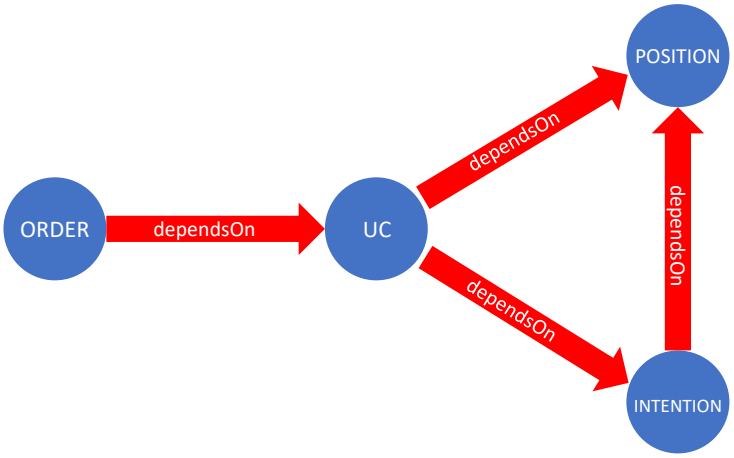
APPROACH We selected projects from the data set proposed by Marquez et al. [149]. Projects had to be developed with microservices. They had to be implemented using Java with Spring to validate the Shared Persistence detection and/or using Docker to build the call graph needed for Cyclic Dependency detection. There were no constraints for the detection of Hard-Coded Endpoints. We first ran Arcan on all projects coming from Marquez work (30 projects), detecting possible microservice smells. We identified 5 projects affected by at least one type of smell (Table 6.7).

Then we validated the results of the detection strategies by manually inspecting the detection of the smells in the five projects. The manual validation was performed separately by two of the authors; cases of disagreement were discussed. We obtained a precision value of 100% since all the found instances represent true instances of microservice smells. On the other hand, we were not able to compute the recall value due to the lack of detailed project documentation and project developers feedback. We aim to extend our validation by analyzing more open-source or industrial projects, possibly with the assistance of developers and information about known microservices defects.

11 YAML is a human-friendly data serialization standard. https://yaml.org/, accessed October 2021

Table 6.7: Analyzed projects

Name	# Services	Link	Lang
Micro-company	4	github.com/idugalic/micro-company	J
Service Commerce	8	github.com/antonio94js/servicecommerce.git	JS
Sharebike	4	github.com/JoeCao/qbike.git	J
Sitewhere	19	github.com/sitewhere/sitewhere.git	J
Task Track Support	5	github.com/yun19830206/CloudShop-MicroService-Architecture.git	J



The call graph shows microservices as nodes and microservices dependencies as edges. The *Sharebike* project is a platform for renting and sharing electric vehicles. The **Order** server manages vehicle requests and trip information; the **UC** (User Center) manages user registration; **Position** handles vehicle discovery and user trip history; **Intention** manages the match between user and nearby vehicles.

VALIDATION RESULTS Out of the five selected projects, Arcan was able to detect 2 instances of the Shared Persistence smell and 6 instances of the Hard-Coded Endpoints smell. Regarding Cyclic Dependency, the projects under analysis were not affected by this smell. However, Arcan was able to create the dependency graph of the Java projects developed with Spring and using Feign, and store it in a Neo4j graph database: Figure 6.3 shows the example of the *Sharebike* project. The image was captured by the Neo4j browser.

SHARED PERSISTENCE RESULTS We detected Shared Persistence smells in two projects: *Micro-company* and *Sharebike*. Table 6.8 shows the details of the detected smell instances. As for the Micro-company project, the data is shared by two different services that access the same Mongo database [166], named "blogpost". This information can be found in the configuration files of the two services, stored in the configuration service of the application. The Mongo database allows isolating data through the *collection* feature, where a collection is a set

of stored document. A set of collections makes up a database, where collections can be accessed by specific services. However, this is not the case for Micro-company because all services access the same collection. In the case of the ShareBike project, the smell affects all the services. In particular, every configuration file (*application.yml*) contains the declaration of the same mysql database, whose name is "qbike". From the manual validation we obtained a precision of 100%, since all the found instances represent true instances of microservice smells. On the other hand, we were not able to compute the recall value due to the lack of detailed project documentation and expert support.

Table 6.8: Shared Persistence results

Db Type	Affected Services	Shared Database Name
Micro-company		
mongoDB	query-side-blog	blogposts
	command-side-blog	
Sharebike		
mysql	<all>	qbike

HARD-CODED ENDPOINTS This smell was detected in the projects *Task Trak Support*, *Sitewhere*, and *Service Commerce*. Table 6.9 shows for every affected Java class the detected IPs and ports and the description of the smell instance. Concerning the *Task Track Support* project, the *DictRequestUrl* class contains a set of http requests such as `http://192.168.1.108:9003/api/gateway/getOrderByUserIdNormal`. The *CuratorFrameworkFactoryBean* Java class contains the address to the ZooKeeper service [16], which is used for maintaining configuration information. The last two rows of the table report hard-coded http requests in two test classes. As regards Sitewhere, the tool detected one occurrence of the smell, in the class *EventSourceTests* with the value 0.0.0.0:1234, caused by the presence of the configuration of a test server. Considering the JavaScript project, *Service Commerce*, the tool detected one occurrence of the smell in a file containing an http URL. As in the case of Shared Persistence, all the smell instances found represent true instances of microservice smells, therefore the precision obtained is 100%. In this case, too, it was not possible to calculate recall due to the lack of detailed project documentation and expert support.

FINAL REMARKS The application of Arcan to the selected projects demonstrates that it is possible to automatically detect smells in projects, and therefore proves that we can propose our implementation to practitioners and researchers. The lack of access to industrial projects de-

Table 6.9: Hard-Coded Endpoints results

File name	Matched ip:port	Description
Task Track Support		
DictRequestUrl.java	192.168.1.108:9003	hard-coded multiple times in the list of all the http requests that can be made versus the *API gateway*
CuratorFrameworkFactoryBean.java	127.0.0.1:2181	zookeper address
CloudShopRequestConcurrentTest.java	192.168.1.108:9002	hard-coded in an http request to the *order* API
RequestTestMainApp.java	192.168.4.181:9002	hard-coded in an http request to the *order* API
Sitewhere		
EventSourceTests.java	0.0.0.0:1234	test server configuration
Service Commerce		
mercadopago.js	190.207.117.222:3000	hard-coded assignation of URLl to a service

veloped with microservices, and the lack of documentation on existing open-source projects did not allow us to compute the recall. We aim to extend our validation by analyzing more open-source or industrial projects, possibly with the support of developers and information about known microservices defects.

6.2.3 *Micorservices smells identification - Aroma*

AROMA is based on dynamic analysis. Tracing information is collected on each host by instrumenting them with Zipkin and then stored in a Json file. We chose to integrate AROMA with Zipkin, instead of developing our tracing system, because we preferred to reuse an existing tracing tool with a mature community[12]. The outcome of of AROMA is the *Microservices call graph*, which represents services as nodes and services dependencies as edges. Examples of analysed projects and graphs can be found on the AROMA Gitlab page[13].

The Json file generated by the execution of Zipkin contains a set of *span*: the information collected during a specific remote activity (e.g., a Remote Procedure Call or messaging producers and consumers) by a single host. In our case, spans contain data about client requests, microservices responses and endpoints from the point of view of each single microservice. A Zipkin *trace* is a series of spans which form a *latency tree* which provides an overview of the path a request takes through the entire system. Thanks to spans, we are able to understand which are the services of the architecture, which endpoints

12 https://zipkin.io/pages/community.html, accessed October 2021
13 https://gitlab.com/essere.lab.public/aroma

they expose and which relationships they actually have one versus the others.

We use a representation named *microservices call graph* to store as many information as needed to detect the MS smells. We exploit the Tinkerpop graph framework, as we already did for other Arcan, since it allows to both build and then query the call graph and also offers the drivers for many graph databases. Tinkerpop allows to define Vertex (node) and Edge objects, and both can have properties. Nodes in AROMA can be of two types: service or endpoint. Each service node stores a set of properties related to the Zipkin traces: *parentId*, *timeUTC* and *duration*. Moreover, they also have a *serviceName*, indicating the name of the microservice. The other node type, endpoint, stores network information such as the *ipv4* address, the service *port* and the *type* of endpoint, local or remote. Each service can have one or more associated endpoints, and this information is stored as an edge with label "has" from a service node towards an endpoint node. Service dependencies, which we represent if the Zipkin traces recorded some kind of communication, are stored as edges with label "call". This kind of edges stores an additional information, the *weight*, i.e., the number of requests recorded during a single execution of the tool. Also Zipkin provides a graph to inspect the results of its monitoring. However, we decided not to directly exploit the Zipkin graph to enable the future extensions of the AROMA, e.g., extend the graph with information about network protocols details, message brokers and containers. The Tinkerpop schema is open by design to further extensions, and the set of displayed information can be enlarged to represent other aspects concerning MS architecture, e.g., network protocols details, message brokers and containers.

Following, the definition and detection strategies of the considered smells.

MEGASERVICE *Definition:* a Megaservice is a service that does *a lot of things* and looks more like a monolith than a microservice [236]. The problem consists in having several business processes implemented in the same service, which is against the principles behind MS design [208]. *Detection:* we start from the assumption that if a service receives many requests from the client, with respect to the request load of the other services, then it could be an example of megaservice. We propose to identify it by *a)* storing on the call graph the number of requests received (*weight*) during a test, i.e., during a single execution of the system; *b)* Check for each service whether the weight overcomes a given threshold: if true, the service is affected by the smell. The threshold corresponds to the median value of the distribution of weight in the analysed project. Setting thresholds is not a trivial task: our approach consists in setting thresholds through the adaptive threshold method [18], by adapting its value depending on

the total number of exchanged requests in the system. Such services are composed by several modules, and developed by several developers, or even several teams.

CYCLIC DEPENDENCY - DYNAMIC *Definition:* Cyclic Dependency arises when two or more services depend on each other. For dependency in the dynamic case we mean an API call between services [236]. One way to avoid the appearance of this smell is the adoption of the API Gateway pattern [208], leaving the role of mediator between the different services to the gateway.

Detection: we already proposed the detection of this smell trough the Depht First Search (DFS) algorithm (see Section 6.2.1). However, the previous approach limitation was its dependency towards specific technology stacks: the code had to be written in Java with Spring, or with Feign. Moreover, we recovered static dependencies, without actually running the tool and potentially losing many dependencies (e.g., the ones from the API gateway). With AROMA we can run DFS on edges created by API calls, only considering actual dependencies and improving the detection.

LACK OF API GATEWAY *Definition:* this smell appears when MS communicate directly with each other. In the worst case, the client also communicates directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance. This smell represents the violation of the corresponding MS pattern defined by Richardson [208], named "API Gateway", which suggests that services should not be directly exposed to the outside, but should be hidden behind a message routing service. The presence of an API Gateway allows the mitigation of the requests flow coming from the client, while the internal communication, from gateway to services and also among services, is managed by the system. The presence of this smell can occur in two different scenarios: the API Gateway is absent (not present by design) or the API Gateway is present, but bypassed by some client requests [233].

Detection: Depending on the two scenarios, we identify the complete lack of gateway if *a)* the root node (representing the hypothetical client) has more than 1 outgoing edge, i.e., the client makes requests directly towards the microservices *b)* there is no service containing the term "gateway". If both *a)* and *b)* are not satisfied, then the Gateway is absent. In the case there is a service satisfying condition *a)* and *b)*, we mark a dependency as "bypass" if the dependency has as target a microservice and as source a node different from the API Gateway.

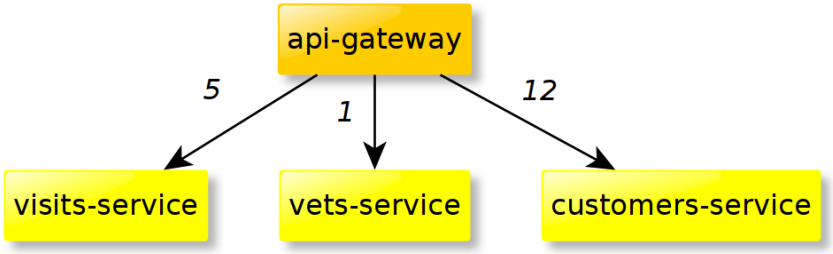


Figure 6.4: Spring PetClinic microservices - Call graph

6.2.4 *Validation - AROMA*

We ran AROMA on three Open-Source projects available on Github: *Spring-PetClinic-microservices*, *LAB Insurance Sales Portal* and *BookStore-App*. We found evidence of microservices smells only in one of them (*BookStoreApp*). Thus, in order to showcase the AROMA features, we also provide an additional analysis report produced from a synthetic example.

Given that Aroma requires the usage of Zipkin, we searched for already instrumented projects, to speed our validation. We had to execute and explore the projects' APIs in order to get the necessary information. This requires to build different environments for different projects (e.g., with Docker or Kubernetes) and build the projects themselves. This was not a trivial task, since there are no public repositories, according to our knowledge, available to test and validate tools for MS quality assessment, with out-of-the-box examples. We searched projects on Github, by filtering with the microservices and Zipkin tags, and selecting the most starred (popular) ones. We provide the execution traces along with the list of tested APIs for each project in our replication package[14].

The instructions on how to run the examples and replicate the results are in the AROMA Gitlab repository[15].

SPRING-PETCLINIC-MICROSERVICES is the distributed version of the Spring PetClinic sample application, developed to support the learning and usage of the Spring framework[16]. Its API is instrumented through the use of openZipkin [184], making it a convenient example to test our tool. Figure 6.4 shows the generated call graph[17]. Thanks to the Zipkin traces, our tool recovered all the three backend services and also the API gateway. Moreover, it stored the number of requests

14 https://drive.google.com/drive/folders/1iPhWoulSXkxtz9kfzMb4JkvtfCsG0154?usp=sharing
15 https://gitlab.com/essere.lab.public/aroma
16 https://github.com/spring-petclinic/spring-petclinic-microservices, accessed October 2021
17 Picture taken with yEd [263]

received by each service. Notice that one of the nodes, customers-service, is dispatched with more requests than the other services, i.e., the edge which connects the gateway to this service has weight equals to 12. By checking the documentation[18], this is justified by the fact that the service addresses both the management of the customer entity and the pet entity. Thanks to trace analysis, we are able to identify services requests and to detect the services which maximise this attribute. In the case of PetClinic, this means identifying the "biggest" services in the architecture. In a meaningful system (i.e., not a toy example), when the biggest service is stressed with many requests, it results in an instance of the *Megaservice* smell. Unfortunately, as previously outlined, this project is very small and in this case the choice of managing the two concepts (customer and pet) inside the same microservice is justifiable and does not result in a maintaibility issue. However, we report this example in order to showcase the tool and the results it provides. In brief, we did not find the Lack of API Gateway smell, but we identified an instance of Megaservice, affecting customer-service.

LAB INSURANCE SALES PORTAL LAB Insurance Sales Portal[19] is a project developed by Alktom Lab, a simple insurance sales system designed with a microservice architecture using the Micronaut framework[20]. Figure 6.5 shows the call graph of the project obtained through our AROMA tool. As we can be see, AROMA identifies six microservices, the API Gateway and also *Consul*, a discovery service [57], used to enable the other services to discover each other by storing location information (like IP addresses) in a single registry. This is why all dependencies point to this service, because it acts as a proxy. Even if being able to detect the Consul service is useful in terms of completeness, it hides the actual interaction among the microservices. Also Granchelli et. al faced this problem [96] while developing their tool: they implemented a solution where the software architect can indicate, after the reconstruction of the architecture, which services are actually discovery services, through a graphical editor. We aim in the future to automatise the detection of discovery services, to improve the generation of the call graph. Concerning the detection of smells, we did not found instances in this project.

BOOKSTOREAPP This project is an Ecommerce project[21] where users can buy books. The application has been developed using Java, Spring

18 https://github.com/spring-petclinic/spring-petclinic-microservices, accessed October 2021

19 https://github.com/asc-lab/micronaut-microservices-poc, accessed October 2021

20 https://micronaut.io/, accessed October 2021

21 https://github.com/devdcores/BookStoreApp-Distributed-Application, accessed October 2021

and React. Figure 6.6 shows the resulting call graph. As we can see, we have cases of Lack of API Gateway, specifically of the first scenario: the client (`user-side`) has a direct dependency (can make requests) towards all the MS without passing from `bookstore-api-gateway-service` (the API Gateway).

SYNTHETIC EXAMPLE Given the few available projects for the validation, we produce a synthetic collection of Zipkin traces to be feed to AROMA. The traces simulate a MS architecture affected by two smells. Figure 6.7 shows the resulting call graph. The represented example is a Zipkin v2 standard reproduction, containing a collection of traces from an API query. We reproduced a "real" execution of a microservices system manually instrumented with Zipkin, by taking into account the mandatory and self-generated fields, such as the *id* of the traces and the associated spans. We also associated to each span a "serviceName" consisting of one letter of the alphabet. This synthetic study therefore offers the possibility both to analyse a single smell in detail and to study its multiple occurrences. In this example it is possible to notice that there is a bypass of the API Gateway (`client` towards service `D`) and an instance of Megaservice (service F is stressed with many requests, compared to the other services).

6.2.5 *Final Remarks*

We are aware of the limitations of the validation of both Arcan extension and AROMA. We analysed few projects, however, it is not easy to find this kind of projects, since there are no public repositories, according to our knowledge, available to test and validate tools for MS quality assessment, with out-of-the-box examples. We aim to validate the tool also on industrial projects to collect feedback from developers. Concerning AROMA, we also suffer of the intrinsic prob-

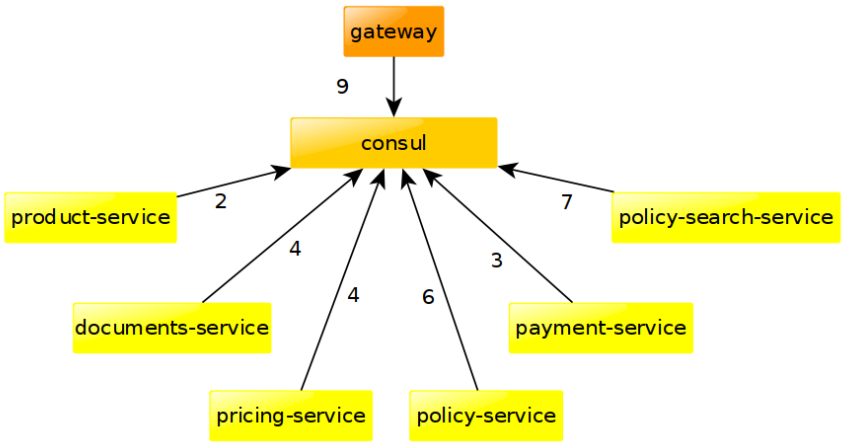


Figure 6.5: LAB Insurance Sales Portal - Call graph

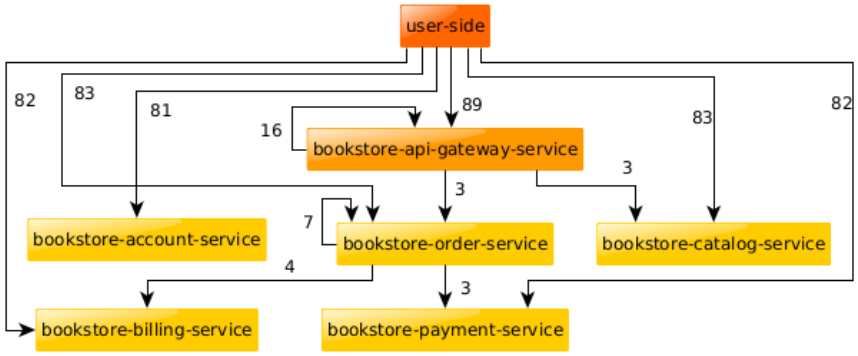


Figure 6.6: BookStore - Call graph

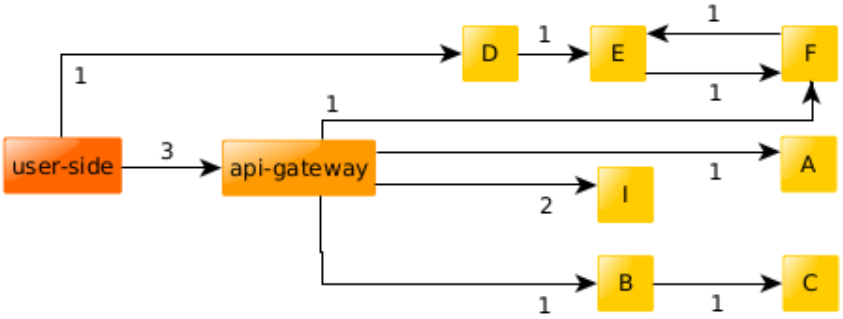


Figure 6.7: Synthetic example - Call graph

lem of dynamic analysis, i.e, the coverage of the tests used to run the analysis might not be satisfying. However, to enable reproducibility, we provide the execution traces along with the list of tested APIs for each project in our replication package[22].

22 https://drive.google.com/drive/folders/1iPhWoulSXkxtz9kfzMb4JkvtfCsG0154?usp=sharing, accessed October 2021

6.3 SUMMARY OF THE FINDINGS

Along with the studies on monolithic architectures, we started exploring the architecture erosion of microservices too. This research line offers many challenges for software analysis tools developers and AS researchers, due to the change of architectural paradigm: the technological heterogeneity of services, their distributed nature, the lack of golden standards for the validation of analysis approaches, among the others.

We started exploring this research field by analysing the migration process from Java, monolithical projects towards microservices. We conducted two case studies in two different industries, with the aim to investigate whether the presence of AS in the monolithical project hinders the extraction of self-contained services and also to validate the Arcan extension for microservices candidate identification. From our results, we acknowledged that AS slow down the migration process. In particular, smells related to dependency issues (CD, UD and HL) make difficult to define the microservice boundaries, i.e., the classes and packages which should be included into the single service. For instance, having a Cyclic Dependency between two packages logically belonging to two different microservices forces the developers to refactor the architecture by relocating pieces of code from one package to the other, or to duplicate pieces of code in order to decouple the packages.

Also smells which impact the separation of concerns principle (SF and FC) are a problem for the migration. Developers from our case studies pointed out that such smells indicate the presence of concerns which should become a service, but at the moment are dispersed all over the project. This means that, in order to extract a cohesive service, developers must identify every dispersed piece of concern and move it into the correct place. Developers underlined also that tools like Arcan are useful in such cases, because they indicate where the concerns are scattered. Indeed, Arcan was successfully adopted in both case studies to obtain the list of candidate microservices to migrate.

A further step we made in this research field was to implement a tool for microservices smells detection, the counterpart of architectural smells in such architectures. We first proposed an extension of Arcan, to statically detect three microservices smells, and subsequently refined our approach by developing AROMA, based on dynamic analysis, detecting two additional types of smells.

Our approaches are at the first stages of development and are validated on small, Open-Source systems. We aim in the future to extend them, especially the one implemented in AROMA, reinforcing the model of the reconstructed microservice architecture with more details, such as information about the infrastructure, and by adding the

detection of additional smells. Additional future directions concerning this subject can be found at the end of the thesis in Section 8.2.

7 RELATED WORK

In this Section we introduce some related works concerning the various research lines we addressed in this thesis, related to architectural smells and architectural debt. In particular, we describe works from the literature concerning:

- AS detection techniques and tools;
- Empirical studies on AS, i.e., studies aiming to analyse the behaviour and characteristics of smells starting from the data of a large number of Open-Source or industrial projects;
- Architectural debt evaluation works, concerning approaches developed to identify ATD. We also introduce some empirical studies about (generic) technical debt indexes, since there are few empirical analysis of the architectural ones;
- Case studies about the migration to microservices and tools for microservices reconstruction and microservices smells detection.

AS are a relatively new concept, without a fully established terminology: they are also called *antipatterns* [121], *design smells* [89], or *architectural flaws* [163]. Some authors use these terms interchangeably, while others propose more complex ontologies, e.g., antipatterns can be considered as a subtype of architectural smells by some authors, and some of them, like Tangle or Hub, are called antipatterns [121] or architectural smells [90, 231]. Concerning the term *design smells*, Sharma et al. actually distinguish design smells from architectural smells. However, some of their design smells correspond to some of our AS, but at class level. We can map what they call design/architectural smells to our distinction class/package smells.

7.1 ARCHITECTURAL SMELL DETECTION AND PRIORITIZATION

Under the AS detection works expression we gather the literature which addresses techniques, frameworks and automatic tools for AS identification. We divide these related works in three parts: works concerning the detection of dependency issues-based AS, works introducing Natural Language Processing approaches for AS detection and finally some approaches for AS prioritisation.

7.1.1 *Tools and data structures for the detection of dependencies issues-based AS*

We now provide and overview about techniques and tools for the identification of AS related to dependency issues, namely Unstable Dependency (UD), Hub-Like Dependency (HL) and Cyclic Dependency (CD). Notice that, apart from few exceptions, the majority of academic and industrial tools focus on the detection of CD.

In general, the majority of AS detection techniques rely on metrics computation. A software metric is a function which measures a property of the code. Such approaches consist in collecting various software metrics, such as Lack of Cohesion in Methods (LCOM) and Coupling Between Objects (CBO) [53], and then set logical rules and thresholds to detect AS. One example is the work of Moha et al. [164], who developed a tool named DECOR to detect design smells. This is a kind of approach inherited from the code smell detection literature, where the compliance of small snippets of code are checked against logical rules [128].

The limitation of such approaches lies in the inability of identify smells which affect the structural dependencies of software systems. Such smells do not manifest themselves as an anomaly with respect to a defined rule, but appear as structural *patterns*, i.e., perceptual structures which involves architectural components and their dependencies. That is why in the literature many approaches for AS detection rely on the abstract representation of the software architecture through models: first, the tool reconstructs the software architecture by representing it with a suitable data structure (the model), then the tool run detection algorithms on the model.

Examples of data structures for software architecture representation are the *Design Structure Matrix* (DSM) [71], the *Design Rule Hierarchy* (DRH) [107] and the *Dependency Graph*. In the case of DSMs, the architecture under analysis is represented as a square matrix, where architectural components are labelled in the rows and in the columns and the values in the cells indicate the number of dependencies among the different components. Examples of tools based on DSM are Designite [220], Lattix [130], Structure101 [103], Sonargraph [270].

Concerning DRHs, the name *design rule* refers to Baldwin and Clark's design rule theory [29], where design rules are the key architectural decisions that decouple the rest of the system into independent modules that can be implemented, revised, or replaced without influencing other parts of the system. The DRH algorithm takes as input a DSM and clusters source files so that their architectural roles can be made explicit, i.e., to manifest the differing architectural importance of different source files. A DSM, clustered using the DRH algorithm, has three key features. *1)* the design rules and modules are organised

in a hierarchical structure, with design rules on the upper layers and the modules decoupled by the design rules on lower layers; *2)* the modules in lower layers depend on the modules in higher layers, but not vice-versa; *3)* modules in the same level are mutually independent from each other. Cai et al. [49] refined the DHR algorithm and conceived the Design Rule Space (DRS) model, on the top of which the DV8 [48] tool is developed.

Finally, a *dependency graph* is a graph model where architectural components are represented as nodes and architectural dependencies as edges. The tool discussed in this thesis, Arcan, is based on dependency graphs (see Section 2.1), that we prefer because it enables the exploitation of graph algorithms for AS detection. Other examples of tools which exploit dependency graphs are Dependency Finder [63], JArchitect [110], ClassCycle [55] and NDepend [170]. Finally, AI Reviewer [6] proposes a custom model called *Core Analysis Model* (CAM), a detailed abstract representation of the program source code which can represent program entities and their relationship.

All the indicated tools focus essentially on the detection of Cyclic Dependency smell, while AI Reviewer, Arcade, Designite, Arcan and DV8 can detect more than one AS. See Section 2.5 for a more detailed comparison of Arcan algorithms with other tools' strategies.

We have briefly outlined above some tools for AS detection, while a catalogue of AS detected by tools has been proposed by Azadi et al. [27], where they report the detailed description of the detected smells and the corresponding violated design principles.

7.1.2 *Natural Language Processing models for the detection of separation of concerns-based AS*

Other approaches exploit techniques coming from the information retrieval and the Natural Language Processing field, in particular to model software concerns (see Section 2.1).

Exploiting NLP and Machine Learning (ML) techniques to extract semantic information from software gained popularity in the last 10 years. We now first introduce some examples of models leveraging software semantic for software recovery and then we focus on the approaches aiming to detect AS.

Jalali et al. [212] propose a multi-objective fitness function, named MOF, which exploits both structural and semantic features (such as semantic contained in the code comments and identifier names), to automatically guide optimization algorithms to find a good decomposition of software systems. They claim that the provided decomposition is more understandable and meaningful respect to the ones obtained by exploiting only structural information of the software. Boaye et al. [32] propose a search-based approach that uses structural and lexical information to recover the layered architecture, at

the package level, of an Object-Oriented (OO) system. Lexical information consists of significant keywords derived from identifier names (i.e., names of packages, classes, methods, fields and parameters) and comments found in the source code. They assume that two packages are conceptually related if the packages' lexical information is similar. The authors model software concerns as topics extracted by using LDA: the conceptual relationship between two packages is computed as the cosine similarity between their corresponding topic proportion vectors.

Corazza et al. [58] consider the source code of Open-Source software divided in six "zones", depending on the granularity level (class name, method name, attribute name, parameter name, comment, source code statement) and try to assign to those zones an estimation of their relevance based on the contained lexical information to improve the quality of software recovery. They define a probabilistic model of the lexemes distribution and then exploit it to compute similarities among source code classes, which are then grouped by a k-Medoid clustering algorithm.

The presented models and methods aim to recover software architecture by considering semantic features of software itself. The following works go further by investigating the usage of semantic information to detect AS in Open-Source software. Diaz-Pace et al. [66] explore whether social network analysis is useful to extract information from a software architecture in order to predict new dependencies and possible future appearance of AS in Java projects. They model software architecture as a dependency graph where edges are enriched with topological and content-based information modeled as *bag-of-words*. In particular, they propose to consider Java classes and packages as a bag-of-words containing the most representative tokens that characterize their source code, such as identifiers, methods' names and comments. Then, they compute edge information as the Cosine Similarity scores for the different bag-of-words representations.

Garcia et al [92] propose to recover software systems thanks to the detection of software concerns. To obtain concerns, they exploit a statistical language model named Latent Dirichlet Allocation (LDA). Their approach is implemented in a tool named Arcade, which is also able to detect Scattered Functionality and Feature Concentration in Java projects [133]. Their detection approach is based on the number of topics assigned to each software component.

The limitation of all the above mentioned approaches is that they do not consider context information of the analyzed code, moreover they were not designed for source code analysis. Regarding this aspect, a study from Hellendoorn et al. [104] suggested that existing neural networks, born specifically to model natural language, are not the best solution to represent code semantic. Instead, simpler mod-

els, such as *counting models* taking into account only the frequency of a term, perform better than deep learning models and require also little computational time. From 2017, when the paper was written, many other deep learning models were developed, with a focus on representing the semantic of code. In particular, many works were developed for software security tasks [73][267] [265] and code summarisation [257][47][252]. For what concerns our experience, we experimented with code2vec [12], a neural model which is able to predict semantic properties of given snippets of code (see Section 2.4). From our study, we acknowledged that the usage of such model requires a great amount of resources, in terms of computational capability and time. That is why we introduce a new technique (see Section 2.1), based on the tf-idf metrics, to model software concerns. We merge in a unique data structure the information on architectural dependencies and architectural concerns and exploit it to detect two AS which violates the separation of concerns principle (Scattered Functionality and Feature Concentration). In this way, we can identify them not only by exploiting structural information (as done in Designite [231]) but also semantic information. Moreover, this approach overcomes the problem of fixing the parameters of concerns detection required by the topic models' algorithms (as done in Arcade [133] and also by us, see Section 6.1.1), since the tf-idf computation is completely automatic and requires far less resources with respect to a neural model.

7.1.3 *Architectural smells prioritization and criticality evaluation*

We now outline some related works done in the literature on the evaluation of criticality and prioritization of code and architectural smells. What distinguishes the following works is the kind of information used to estimate the priority of a smell.

For instance, concerning code smells, Vidal et al. [215] presented an approach to identify the most critical smells based on a combination of three criteria, namely: past component modifications, important modifiability scenarios for the system and relevance of the kind of smell. Also Rani et al. [205] proposed a methodology for code smell prioritization. First, it detects smelly classes using structural information of source code, then mines change history, as done by Vidal et al., to prioritize the smells. Always according to code smells, Sae Lim et al. [213] exploited the *developers' context* (a list of issues extracted from an issue tracking system) to define priority. Instead, Arcelli et al.[83] proposed a severity index of the smells based on how the metric thresholds used for the smells detection are exceeded. Similarly, Guggulothu et al. [99] proposed a prioritisation approach for four code smells (Long Method, Feature Envy, God Class and Data Class), depending on their impact on design quality, where the impact is

measured depending on the overcome of a set of metrics such as coupling, size, complexity and cohesion.

More recently, Pecorelli [190] proposed a machine learning approach to prioritise the application of refactoring on code smells. They generated a rank of code smells according to the perceived criticality that developers assign to them.

According to architectural smells, there are fewer studies about prioritization. Martini et al. [155], performed a study on the analysis of the most critical AS through the feedback of the developers of two industrial projects. The smells have top refactoring priority in the opinion of practitioners are the ones with the highest negative impact on the maintainability and evolvability of the project. On the same line, Oliveira et al. [182] investigated criteria that developers use in practice to prioritize design-relevant smelly elements with the aim to develop a set of prioritization heuristics. From their results, two out of nine heuristics reached an average precision higher than 75%. Finally, Vidal et al. [249] presented and evaluated a set of five criteria for ranking groups of code smells as indicators of architectural problems in evolving systems.

According to our knowledge no extensive work has been previously done on the analysis of the evolution and correlation between criticality and cost-solving, evaluated in terms of PageRank of AS and Severity metrics. In this thesis we introduced a work (see Section 2.2.2) on the evolution of a set of projects (10 projects, 22 versions each, for a total of 264 versions), and we analyzed the correlation existing between the two metrics through Spearman and Kendall correlation tests. Moreover, we studied the evolution of the metrics in the project history. Finally, we proposed to exploit PageRank as a proxy for criticality, and Severity as a metric to estimate cost-solving.

7.2 EMPIRICAL STUDIES ON ARCHITECTURAL SMELLS

The hypothesis of increased change- and defect-proneness resulting from the presence of AS was observed by Mo et al. [163], who proposed five file-level architectural flaws and found their significant correlation with error-proneness and change-proneness. Similarly, Oyetoyan [185] identified a positive correlation between the presence of Cyclic Dependency AS and the change frequency.

Sharma et al. [221] conducted an empirical study to investigate the relationship between design smells and AS in C# projects. They studied correlation to check whether, given pairs of design and architectural smells which capture the same concept at different granularities, one of the two is superfluous. They studied collocation, trying to understand if certain types of design smells may act as indicators for specific AS; they also studied causation, by investigating temporal relationship between design and AS to figure out whether some

types of smells cause the others. Thanks to their analysis, they found evidence of the individuality and uniqueness of design respect to AS. In this thesis, we also studies AS correlation, however, differently from Sharma, we focus on the possible relationship among different types of AS at the *same* granularity level. We are interested in finding whether two or more smells of different types, concerning different design principle violations, are correlated or collocated. To do so, in addition to the computation of correlation coefficients, we run the Partial Component Analysis and the association rules extraction, which allowed to investigate relationship among more than one smell types at the same moment.

Aversano et al. [24] studied the evolution of design smells in 8 Open Source Java projects. They discovered that classes affected by design smells are more subject to change, especially when multiple smells are detected in the same classes. They also investigated whether the generic refactoring of the architecture conducted by the developers led accidentally to the removal of the (unknown) smells, without finding positive evidences.

Other authors have analysed the impact of AS on maintainability. For instance, Le et al. [133] analyzed the relationships between AS and issues reported in *issue trackers* (e.g., Jira). They found that AS have tangible negative consequences, resulting in implementation issues and increased maintenance effort.

Herold [105] investigated the relationship between the presence of AS and manually validated *architectural violations*. He identified the links for Unstable Dependencies and for Hub-Like Dependencies AS, but with small effect sizes. He concluded that the presence of AS cannot alone explain erosion of architecture, but it does play a contributory role. Brunet et al. [45] also studied the evolution of architectural violations in 76 versions of four systems, by comparing the intended and recovered architectures of a system. They found that architectural violations tend to intensify as software evolves, and usually *a few design entities are responsible for the majority of violations.*

Architectural smells impact different aspects concerning maintainability, including system *understanding*. The impact of AS on understanding was highlighted in an empirical study by Abbes et al. [2]. They found that the presence of several antipatterns in one code entity significantly impedes the programming performance of developers.

The comprehensive and diverse nature of AS has also been the subject of investigation. Sas et al. [216] analysed the evolution of three AS and their characteristics in 524 versions across 14 different projects. They focused on growth rate, on the importance in the system of the elements affected by the smell over time, and on the time each smell instance affects the system. They found out that the different types of

AS evolve differently. In our work, we also analyse smells' evolution: however, we consider three additional AS respect to Sas work.

Switching to code smell relationship with AS, Arcelli et al. [78] presented a study on possible correlations between AS and code smells. They appeared to be linked only in a few cases. Therefore, the presence of AS cannot be inferred from CS and AS require separate methods for dealing with them.

Finally, we [17] developed various machine learning models trained on multiple versions of four Java projects. We found that the presence of AS in a previous versions of the system also affects them in the future.

The presented works indicate that the presence of AS affects various important properties and quality characteristics of software systems. In this thesis, we analysed the relationship between AS and Design Patterns, and also the possible correlations among AS of different types.

7.3 ARCHITECTURAL DEBT EVALUATION

We addressed the research about ATD in terms of evaluation of the amount of ATD in projects belonging to different applications domains. We introduce here some works about approaches for ATD identification and present also some empirical studies where TD indices are employed. Currently, to the best of our knowledge, there are only two indices for ATD evaluation based on architectural aspects: our ADI and ATDx [247]. Further details about ATDx are reported in Section 5.1, compared to ADI facets.

7.3.1 *Identification of ATD*

When speaking of "ATD management" we refer to those practices aimed at identifying ATD items and/or symptoms and remove them. In the context of this thesis, we focused on ATD identification, and in particular in this Section we introduce some of the existing, proposed approaches to identify ATD.

Li et al.[142] conducted multiple case studies on thirteen open source projects to evaluate the ability of modularity metrics [3] to indicate the presence of ATD. They focused on such metrics because they claimed that ATD should be measured starting from source code, i.e., with metrics calculated on code. In order to verify the ability of their set of metrics to estimate ATD, they test the correlation between the metrics and the average number of modified components per commit (ANMCC), a metric indicator of ATD (a higher ANMCC indicates more ATD in a software system). From their results, it appears that two modularity metrics, namely Index of Package Changing Impact (IPCI) and Index of Package Goal Focus (IPGF), have significant cor-

relation with ANMCC, and therefore can be used as alternative ATD indicators.

Kazman et al. [116] measure ATD by identifying architecture roots, i.e., flawed architectural structures (set of connected, defective files). In particular, they exploit the Design Rule Space (DRSpace) analysis approach [49] to locate architecture debts in a few clusters of files. Given a project to be analysed, the approach exploits different kinds of data: source file dependencies, (Git) revision history and (Jira) issue history. The information are processed by the authors' tool, Titan, and represented as DRSpaces. After that, they visualize the results on Excel spreadsheets, pointing out to the architects how the architecture flaws propagate errors. After these flaws (named *architecture hotspots*) are confirmed by the project's architects, they extract data from the development process to quantify the penalty these debts are incurring, estimate the potential benefits of refactoring, and make a business case to justify refactoring.

However, not all authors agree about the goodness of source code metrics in evaluating ATD. For instance, Li et al. [141] argue that such ATD identification approaches can only identify source code-related issues (e.g., the modularity violations). They cannot identify ATD caused by architectural decisions that are not reflected in the code, such as inappropriate, immature or obsolete technologies used, and architecture drift [240].

To overcome the limitations of such approaches, they propose to identify ATD by taking into account architectural decisions made during the architecting process and change scenarios. They indicate as Architectural Decision (AD) a design decision that affects the architecture design space for a target software system [124]. A change scenario is a maintenance or evolution task to be performed in the software system. In their view, ATD items are caused by the ADs that negatively impact the change scenarios. They validated their approach by conducting an industrial case study where they asked developers to validate the identified ATD items, i.e., to indicate whether the ATD item actually affects the maintainability and evolvability of the system under analysis. The interviewed developers reported that the approach is useful and easy to use, and it supports release planning and ATD interest measurement.

The approaches introduced until this point are automatic or semi-automatic, and are suitable for the integration into analysis tools. However, ATD identification can also be manual. For instance, Martini et al. propose their own approach to identify and manage ATD. They first conducted a multiple-case study [154] to investigate the factors responsible for the accumulation of ATD and to understand how it evolves over time and filed a taxonomy of 16 factors. Subsequently, they developed and evaluated a method, AnaConDebt[152], for the estimation and prioritization of refactoring ATD items. AnaConDebt

provides indicators that would estimate the important factors responsible for the growth of ATD interest and therefore would warn the organization (including non-technical stakeholders) that the refactoring is urgent. They validated AnaConDebt by analyzing, together with several practitioners, 12 existing cases of Architecture Debt in 6 companies, proving useful to support the architects into systematically analyze and decide upon a case.

Still Martini et al. [156] developed another framework for ATD identification, but in this case they aimed to obtain an automatic solution. While AnaConDebt strongly relied on the developers's actions and feedback to provide results, in this work the author's focus on quantifying ATD in the form of lack of modularization. They propose a Measurement System and an estimation formula that indicate to developers possible candidates for refactoring and support a cost-effective estimation of the ATD interest. Also in this case, they evaluated the approach with a large case study: the results provided evidence that refactoring actions to achieve modularity in a software systems can pay off in terms of development and maintenance effort.

In this thesis, we presented a set of studies concerning the evolution of ATD in two application domains and about the impact the reuse of software components has on the amount of ATD. From our results, we ackowledged that code-related practices, such as bug fixing and small improvements, can help in keeping the ATD under control.

7.3.2 *Empirical studies on technical debt indexes*

Strečanský et al. [230] compared three TD identification techniques: i. Maintainability Index (MI) [179], ii. SIG TD models [183] and iii. SQALE analysis [168]. Considering 17 large open source Python libraries, they compare TD measurements time series in terms of trends in different sets of releases (major, minor, micro). While all methods report generally growing trends of TD over time, MI, SIG TD, and SQALE all report different patterns of TD evolution.

Similarly, Lefever et al. [135] compared the analysis results of DV8, Structure101 and SonarQube, finding that there is a strong lack of consensus among these TD tools in terms of software metrics computation (same metrics, different results) and TD-affected architectural components identification.

Amanatidis et al. [13] compared CAST, Squore, and SonarQube TD estimation. The findings of the inter-rater agreement analysis suggest that there is a statistically significant and strong agreement among the three TD tools on the measurement of TD at class level. However, a substantial degree of disagreement has also been observed for the measured TD level for numerous classes.

Digkas et al. [67] investigated the relationship between TD, measured with the SonarQube TD index, and what they call *clean new*

code, i.e., code whose TD density is kept below the system average. In particular they performed a large-scale case study on 27 Open-Source Apache projects and found some hints indicating that writing clean new code can be an efficient strategy for reducing TD.

Still Digkas et al. [68] studied the evolution of TD measured with SonarQube in 60 Java Apache projects, and found that, even if the project size, the number of SonarQube issues and the value of complexity metrics of the project tend to increase along time, the normalised (with respect to the project size) TD tend to decrease.

Tan et al. [237] ran SonarQube on a large number of commits of 20 Open Source Python projects to investigate self-fixed TD. The results show that in general the number of self-fixed TD items is small with the respect to the total number of items and that as long as a (Python) project evolves,i.e., by growing in size and number of developers, the number of self-fixed TD items decreases. Moreover, they considered also the type of TD, finding that Test Debt and Design Debt items are likely to be fixed by other developers instead of their original creators.

Still concerning works studying TD via SonarQube, Lenarduzzi et al. conducted some empirical studies about TD [137] and its relationship with bugs [136] and faults [138]: their results revealed that dirty classes might be more prone to change than classes not affected by SonarQube issues, even if the actual differences between the two groups of classes is small and depends also on the type of issue.

Finally, according to the comparison of tools for TD identification and their indexes, we took part in a study listing and comparing tools for technical debt measurement [26]. Given a set of selection criteria, we described 9 TD tools. Our analysis offers practitioners a clearer overview of the current landscape of TD tools and highlights their differences in offered features, popularity, empirical validation, as well as current shortcomings. Our results allow to compare the tools against each other and make an informed choice on which tool best suits the needs of individual developers or their teams.

7.4 ARCHITECTURAL SMELLS IN MICROSERVICES

This section reports studies both concerning the migration to microservices architecture from monolithical architectures and also describes tools for microservices reconstruction and smell detection.

7.4.1 *Migration to microservices*

The discussion on how to migrate from monolithic architectures to microservices produced several practical guidelines to help developers in this process: they usually come from direct experiences in the industry [46], but also from research in academia. We describe below some of the most recent proposed approaches. Balalaie [28] present

a catalog of migration patterns to support the migration from non cloud-native architectures to microservices architectures. Mazlami [160] propose a formal approach to identify components of monolithic applications that can be turned into microservices. Their extraction model represents the system under analysis as a weighted graph on which they run graph clustering algorithms. They introduce three extraction methods which differ in how the edge weights of the graph are computed. Mishra [162] propose an approach to enable the migration from the monolith to microservices by exploiting *data flows analysis*. Their approach exploits the existing data schema joined with other information obtained by using profiling tools to understand the data flow and access patterns: this information is used to propose functional modules, that are candidate microservices. Furda [87] proposes a set of refactoring and architectural pattern-based migration techniques relevant to microservice architectures. Baresi [30] proposes a solution to find the adequate microservices granularity based on the semantic similarity of foreseen/available functionalities described through OpenAPI specifications.

The approaches introduced above do not provide tool support, while in this thesis we introduce a tool to support the microservice migration process (see Section 6.1.1).

On the other hand, Gysel [101] proposed *Service Cutter* which is a method and tool framework for service decomposition based on 16 coupling criteria distilled from the literature and industry experience. The tool is able to extract coupling information from engineering artifacts such as domain models and use cases, represented as an undirected, weighted graph to find and score densely connected clusters. The tool exploits graph clustering algorithms to suggest candidate service cuts which should reduce coupling between services and raise their cohesion. The tool that we introduce in this paper differs from Service Cutter since we collect information on candidate microservices with other techniques such as architectural smell detection and topic detection. Moreover our graph investigation is based on graph algorithms and exploits the information coming from the analysis of the Java bytecode of the project.

For what concerns the detection of architectural smells during the migration process, Carrasco [52] introduced 9 common pitfalls that divides in 5 architectural and 4 migration bad smells. However, they do not offer a tool to automatically identify smells while we propose to exploit the Arcan tool in order to identify possible architectural smells before or during the migration process. In this thesis we provide the description of an Arcan extension, designed specifically for Java projects, able to support the decomposition of the monolithic application allowing to identify the specific Java classes/packages to be considered during the migration process and we describe our experience in using it in two industrial case studies (see Section 6). Our

work is different from the approaches previously described in this section which often are not implemented in a tool.

7.4.2 *Tools for microservice reconstruction and smells detection*

In this thesis, we also introduce two tools for the detection of microservices smells (MS smells), the counterpart of AS in microservices architectures. One of the tool exploits static analysis while the other is based on the parsing of dynamic traces (the execution traces of microservices API calls). Both first reconstruct the microservices architecture and then run detection algorithms to spot the smells.

In this section we report a set of works regarding approaches and tools for the reconstruction of MS and MS smells detection with static and dynamic analysis.

Granchelli at al. [95] developed a tool named MicroArt for the recovering of MS architecture, which combines static code information, by parsing online github repositories, with data collected at runtime, and by parsing execution log files. The limitation of their approach lies in the technological requirements: a project must provide log files and run on docker platform in order to be analysed. The peculiarity of their approach is that they manage to remove from their model the microservices destined to service discovery tasks, to obtain the actual communication flow among services, not masked by proxy services. However, the limitation of their approach lies in the technological requirements: a project must provide log files and run on docker platform in order to be analysed.

Mayer and Weinreich [159] released an approach based on OpenAPI descriptions and runtime data collected via HTTP protocol, intercepting calls through their own custom library. However, this approach is limited to REST compliant services, relying on Spring framework.

Engel et al. [70] created a framework named MAAT which exploits the OpenTracing API to create a MS model and allow to visualize the system's architecture. The framework is also able to compute six metrics [38]. for the evaluation of the quality attributes of a service-oriented system, e.g., coupling, cohesion and granularity. The source code of MAAT is not currently publicly available.

Kleehaus et al. [122] presented an approach named MICROLYZE for the recovery of MS architecture. They describe their approach as *multi-layer*, because it is able to model the business, the application, the hardware layer of MS architecture, and also the corresponding relationship among them. They implemented the approach by relying on existing monitoring tools and by combining the run-time MS data with static built-time information.

Soldani et al. [224] recently proposed a toolchain named μ-TOSCA able to reconstruct the architecture of MS systems, identify MS smells

and support MS refactoring. Their approach mixes both static and dynamic analysis. However, the analysis is limited only to projects deployed on Kubernates[1], because the architecture is derived from the declarative specification of the deployment.

The first tool we proposed [197], for microservices recovery and microservices smell detection, was as extension of our tool for architectural smell detection Arcan. However, we successively developed another tool, named AROMA, which detaches from Arcan and overcomes some limitations, for instance exploits dynamic analysis to enable the identification of a broader set of microservices smells. It is also less subjected to the type of architectures it can analyse, i.e., it can recover microservices developed with different frameworks, languages and platforms. With respect to the the previous works, AROMA *1)* can execute on heterogeneous projects (implemented with many technologies and frameworks), differently, for example, from MicroArt (bounded to Docker platform) and μ-TOSCA (bounded to Kubernates); *2)* is based on existing, popular software (Zipkin libraries) and *3)* is Open-Source and open to continuous extensions, to meet the requirements of developers. The main contribution of AROMA is to provide an open, extensible tool for the reconstruction and the detection of MS smells.

1 Kubernates is a platform for the deployment and management of containerized applications [126]

8

FINAL REMARKS AND FUTURE DEVELOPMENTS

The following chapter concludes the thesis. We first provide a comprehensive discussion which summarises our most relevant results and findings concerning Architectural Smells and Architectural Technical Debt. Directly after, we briefly outline the future works by addressed topic.

8.1 DISCUSSION AND FINAL REMARKS

Architectural smells, as long as architectural debt, are powerful, but yet fuzzy concepts. Every developer experience them, however their perception is not equal for all. This thesis tries to shed the light about the perception and evolution of six architectural smells affecting different design principles. We investigated multiple aspects connected to the AS: How do developers perceive AS, whether they are real problems, how AS manifest in the system, and if they have a relationship with other measurable aspects related to software architecture. From our studies, we acknowledge that there is still a lack of culture about smells and ATD. However, when AS are introduced and are put under the spotlight, developers recognise the AS and recall the difficulties they cause during everyday coding.

We now discuss the most prominent results of our studies by answering some recapitulatory questions.

Which is the most recurring smell in Open-Source Java projects? In total, we analysed over 100 Java projects, some of which also during their development history. We identified AS with our tool, Arcan, and we can conclude that the most present smell in the analysed projects is Cyclic Dependency (CD). Even if this phenomenon is true for all the considered application domains, this type of AS affects the most projects belonging to the graphical editors domain, i.e., Java systems which comprise a Graphical User Interface (GUI). The results of the study on the relationship between AS and Design Patterns (DP) (Section 4.2) gave us even more insights: the GUI components may be particularly affected by CD because, by design, they could require the introduction of *callbacks* or the passing of a self-reference. For instance, imagine a Java class named `Plot` which manages the drawing of a graphic. Such class manages also the creation of related objects, such as `Axes` and `Label`. However, since the objects can be modified by user input (e.g., change labels colour) the `Axes` and `Label` objects may require to notify the `Plot` class that their status changed. Even if this specific case could be managed by implementing the Observer

pattern and consequently by reallocating the classes' responsibilities, developers may not know how to do it and may wrongly add dependencies between classes without realizing that they are introducing CD. In brief, GUI projects are the most affected by CD because, both intentionally or unintentionally, their code is particularly open to the introduction of this type of smell.

What is the perception of developers about AS? What emerged from the case studies we conducted in industry is that developers experience the negative impact on quality caused by AS and perceive them as dangerous phenomena most of the times, even if they do not know the definitions or have never been told about AS. It was not a surprise that the quality attribute that they indicated as "the most impacted" is maintainability. The set of smells we studied leads to concrete problems: for instance having a God Component (GC) makes it difficult for developers to navigate the code and modify the required piece of code and Hub-Like Dependency (HL) is particularly painful in the case of a small change of the central component, because requires the developers to remember each related pieces of code to modify along. We asked them which are the most critical smells, on the base of their experience. Even if CD is the most diffused smell, it is not considered the worse in the practitioners opinion, also because we found many examples of types of CD false positives (existing cycles but with no actual negative consequences), for instance GUI callbacks (a necessary evil). Our aim in future works is defining specific contexts where AS are not problems and enhance Arcan in order to filter AS in such cases. For instance, we could instruct the tool to avoid signaling CD in the case of GUI components in projects belonging to the graphical editors domain.

Concerning the smell's impact on quality, in the practitioners' opinion and experience, HL is the one which impacts the most the system quality and also the smell which gets worse as time passes. An interesting result we had is that also GC is perceived as very dangerous. It is one of the most simple to detect (a large and very complex component can be spot with metrics computation) however the consequences are hard: it leads to the duplication of code (because the existing one is too entangled to decompose and reuse) and hinders the easy turn-over of developers in a team, because explaining what is inside a GC is a time consuming activity. UD, SF and FC are the smells at the end of the ranking. Even if developers experience them and agree with the fact that they are detrimental for software quality, they do not perceive them as the most critical ones, mainly because they have less experience about them.

One of the most prominent challenge that emerged clearly during the PhD work is how can we automatically measure AS criticality. Since the perception of smells varies depending on the personal devel-

oper opinion and uncontrollable external factors, then the criticality of a given instance of smell can change depending on the context.

In this thesis, we proposed two metrics for criticality evaluation, *Severity* and *PageRank*. From our analysis, we found that the two metrics are strongly, positively correlated. Our future aim is to better understand whether this means that two aspects related to AS (cost-solving and criticality) are in accordance, i.e., when the costs to solve an AS are high, also the criticality is high, or instead they capture the same aspect and thus one of the two is redundant. Of course, these metrics are a starting point which aim to provide an automatic, static and repeatable mean to compute the criticality. At the moment, the two metrics only take into account static facets of the smells (centrality and structural complexity). The interesting challenge is to integrate such indicators with additional ones, able to capture also dynamic and organisational (related to developers teams) aspects [9].

What can we say about ATD evolution of Java projects?

We analysed ATD in many different Java systems, of different application domain. The list of the considered coarse-grained domains is: *Libraries and Frameworks*, *IoT Platforms* and *Multi-Agents Systems Platforms*. Concerning Libraries and Frameworks, whom the majority of considered projects belongs to, we analysed many different fine-grained domains: databases, graphic editors, parsers, IDEs, testing tools, middlewares and so on. From our analysis, we acknowledge that the evolution of projects' size (e.g., numbers of architectural components, LOC) and the fact that "the more a projects ages, the more accumulates debt" are not enough to describe the ATD trend. On the contrary, the majority of evolution histories present a lack of trend (i.e., oscillating value of ATD). However, from our manual analysis and the contributions of other authors [116][133], we can claim that the intervention of developers in terms of bug fixing, code cleaning and refactoring is effective in reducing ATD. Even if this conclusion seems obvious, the real result regards the fact that such information about bugs and improvements can be collected from issue trackers, meaning that that this source of information can be exploited to analyse, predict and in general study ATD evolution.

What are the challenges of microservices smells detection? Our research did not stop at monolithical architectures, but spanned across microservices architectural style. In particular, we studied how the presence of AS in a monolithical architecture hinders the migration towards microservices and we developed one tool for dynamic detection of microservices smells, named AROMA, and extended Arcan with the same aim, but exploiting static analysis.

The six studied AS are a problem during migration because they impact software architecture aspects which inevitably must be kept in consideration in the process. The first step when migrating is the

identification of existing components (in the case of a component base architecture) or the identification of cross-component functionalities in layered architectures. This step is necessary to migrate cohesive components (which we call *candidate-microservices*) that will become the actual services in the new architecture. The difficulty comes when the architecture is affected by smells which couple different components among them, for instance CD, when coming in the form of tangles (overlapping of different CD instances) can be very hard to break. However, in the two case studies we conducted in industrial context, we found that SF and FC are the AS which makes the migration difficult the most. This makes sense, because both affect the separation of concerns principle. Having concerns spread across the architecture oblige the developer to manually identify the component boundaries, i.e., to look for the various parts of the service, extract them and at the same time lose a lot of time in refactoring.

We introduced also the concept of microservices smell, the counterpart of AS in microservices. With respect to this subject, we did not conduct empirical studies, however we ran our tools and inspected the first results. The knowledge we got regards the current challenges in developing this type of tools. First, we cannot limit the analysis to static evaluation of code and architecture properties, for the reason that the "dependencies" among different services cannot be identified in the same way we identify Java dependencies. That is why dynamic analysis is needed, to monitor the microservices activity and infer dependencies from execution traces. Other challenges regard the reconstruction of the infrastructure at the base of microservices deployment: also, in this case, the environment in which the services execute is way more complex than the ones of self-contained monolithical applications. Other than the application layer, studying microservices requires knowledge and tools to inspect the network layer, the management of multiple instances allocation, the microservices messaging, the peculiarity of the different platforms, such as Docker and Kubernates, which play a fundamental role in the design of the architecture. In brief, while for monolithical architectures we could allow ourselves to abstract from the implementation choices of an architecture (such as the choice of frameworks, databases, programming language and so on), this is no more true when dealing with microservices. Future works aiming at developing completely automatic tools for microservices smell detection should first investigate and choose the data that we need to acquire in order to effectively reconstruct microservices architecture and build the detectors on the top of such representation.

ALTERNATIVE TECHNIQUES FOR AS DETECTION We did not focus only on static analysis when studying AS detection. We implemented in Arcan a detector able to identify two smells basing on

semantic information, and we are currently developing strategies for AS detection with *dynamic analysis*.

Concerning natural language techniques to collect semantic information, we concluded that (at the moment) simple, term frequency-based methods perform better than approaches employing neural networks. First, it appears that the former better represent the semantic of code, probably because the current neural network approaches were designed for natural (not synthetic) language. Second, the amount of computational resources required by neural models goes over the capability of a basic laptop and thus makes the models unsuitable for integration into detection tools. That is why we integrated into Arcan a strategy based on the tf-idf computation, and we are now able to provide a *semantic map* of the software architecture under analysis (i.e., the feature graph). On the top of the feature graph, we can detect Scattered Functionality and Feature Concentration. We have already presented the approach to practitioners and got some feedback [198], however, in the future we plan to extend the validation of both the feature graph and the detection strategies.

Concerning dynamic analysis for AS detection, we already discussed our approach to identify microservices smells by exploiting trace analysis. However, this is not the only research path we are following. We are currently developing an Arcan extension to detect HL, CD and FC starting from execution traces triggered by integration tests. At the same time, we are conducting a study on eight Open-Source Java projects. We compare the results obtained through dynamic and static-based analysis to understand if dynamic analysis can be successfully used for AS detection. Different from the case of microservices reconstruction, where dynamic analysis is helpful because it allows collecting information about the services communication where the static analysis cannot, in this case we aim to exploit dynamic analysis to discriminate between the AS false positive and true positive instances. The problem is that static analysis has the advantage of being an exhaustive analysis, given that it runs on the entire codebase, but can report false positive instances. Therefore, we conjecture that integrating dynamic analysis could enhance the detection precision of static approaches. For example, let us consider the HL smell. Static analysis collects all the possible dependencies of the component, even if they will never be resolved at runtime, possibly generating false positives. In these cases, dynamically checking the component dependencies could help in reducing the number of false-positive instances.

As done for the feature graph, we merge static information and dynamic ones in a unique graph, the *trace graph*. The strategies to detect the smells are the same adopted in the static case, however, we also consider the number of times the program executed specific parts of code (information we get from traces). The early results show

that dynamic analysis can be useful for architectural smells detection, but in any case, it must be paired with static analysis. This because dynamic analysis has the opposite drawback: if the integration tests used to collect traces are not exhaustive, i.e., have a bad code coverage, then the dynamic analysis could be missing a lot of potential AS.

8.2 FUTURE DEVELOPMENTS

We now briefly outline some future developments of the research themes addressed in this thesis.

ARCHITECTURAL SMELLS DETECTION Arcan currently detects six architectural smells violating three different design principle, in addition to three microservices smells. AROMA detects three microservices smells, by exploiting dynamic analysis. Further developments in this direction could be the definition of detection strategies for other types of smells, both architectural and microservices, and enforce the validation in terms of precision with industrial developers. We plan also to exploit the feedback about the false positive cases of AS to implement a set of filters in Arcan. Filters could include the detection of design patterns (see Section 4.2), to distinguish cases where the smell is introduced by design while programming a pattern; filters could also activate depending on the type of implemented architecture: if it is possible to automatically understand or know whether the architecture is layered, or structured with components, or even designed with microservices, filters could leave out smell instances not relevant for the specific architectural paradigm. A challenging future work, that would be of great use for the entire community, is the creation of a dataset of validated AS to be exploited as benchmark for analysis tool's developers and AS researchers.

EMPIRICAL STUDIES ON ARCHITECTURAL SMELLS

- *A Study on Correlations between Architectural Smells and Design Patterns* The study is open to several extensions. What we described is a first investigation of the relationships between AS and DP. For instance, we discovered a hint regarding the possible correlation between the increase/decrease of AS and the related decrease/increase of DP. If we could demonstrate this correlation, that would mean that projects with a large number of implemented DP are affected by a lower number of AS. In the future, we aim to investigate this relationship further, by also exploring the evolution of the number of AS and DP in the projects with the support of statistical tests. In terms of the association rules, it is clear that Cyclic Dependency (CD) is appears in the majority of them. This fact is not surprising, given that CD is the most frequent smell in the analyzed projects. Thus, it could be interesting to execute the rules extraction once again, by excluding dependencies affected by CD from the dataset. This may lead to the identification of more rules involving different smells with different DP.

- *Architectural Smells Evolution and Correlation: an Empirical Study* For what concerns other future works on the correlation and

evolution of AS, we aim to extend our work on industrial projects. Moreover, we did not consider history-related metrics (code churns, co-evolution) in our study: we aim in the future to investigate the relationship between AS, criticality and such metrics, e.g., whether there is a link between the criticality of AS and the file change frequency. We also aim to further investigate the shared causes that we identified as at the base of the collocation of AS and understand whether/how we can detect them. Finally, we also aim to study the impact of AS on software architecture by analysing the relationship between the presence of AS and other architectural-related issues, e.g., issues mined from issue trackers or other kinds of architectural metrics able to evaluate architecture erosion.

ARCHITECTURAL TECHNICAL DEBT EVALUATION

- *Architectural Debt Index* An interesting future work is the extension of our Architectural Debt Index (ADI) with further AS, along with further studies about AS criticality and prioritization. Moreover, we aim to compare the ADI with other TD indexes, such as the SonarQube index, to investigate the relationship between architectural technical debt measured via AS and technical debt. In particular, we want to compare ADI with ATDx [247], i.e, the other index for ATD evaluation that is currently available (for what we know). ATDx already includes more architectural related facets with respect to Arcan: do such facets contribute with additional information with respect to AS? Do they add noise or confounding factors? These are some examples of questions that we could try to answer.

- *Impact of Opportunistic Reuse Practices to Technical Debt* We studied whether TD is influenced by the adoption of third-party components. In future work, we plan to evaluate more projects and repositories and compare them to programs implemented with other programming languages, like Python. Another research line could include the evaluation of TD and reuse in microservices projects. In general, we also plan to investigate how to reduce the initial search effort of the components to reuse, to enable quicker analysis.

- *Evaluating the Architectural Debt of IoT Projects* We aim to extend the ATD analysis on a larger set of IoT projects, and select them, if possible, depending on how much relevant they are for studying AS impact. Moreover, we are looking for collaborations with companies to acquire the feedback on architectural debt evaluation of developers who works with IoT projects. We aim to extend the analysis by considering other kinds of evolution variables, such as code-churns and the number of changes, to better

understand the relationship between AS and the project evolution. Another interesting research line can be the study of AS which could affect specifically IoT systems, and the development of a dedicated tool for their detection.

- *Evaluating the architectural debt of agent based systems* We found preliminary evidence of the link between bug fixing and the decrease of ATD. Studying the correlation between MAS platform architectural debt and bugs occurrence/fixing could lead to the conclusion that code level bugs have an impact on the accumulation/decrease of ATD. Moreover, the validation of the ATD values found in the projects analysed in the described study could be refined by testing the correlation with issues coming from issue trackers (e.g. Jira [113]). In this way, we could further investigate whether code "Improvements" (which is usually recognized as a category of issues) do have a relationship with the decrease of ATD. Another interesting study could investigate the relationship between ATD and MAS platforms' performance, since a link between performance and design decisions has already been proven. Another next natural step will be to apply this kind of analysis to real MASs developed with these platforms, or to enlarge our analysis to the many add-ons of these four platforms (for example WSIG and OntologyBeanGenerator for Jade [1, 43, 243]), to study if we can identify some common ATDs for MASs or further problems in MAS development platforms.

- *AS Criticality Evaluation* Concerning the support to AS prioritization and the evaluation of AS criticality, we proposed two metrics: Severity and PageRank. We plan to conduct a validation of both metrics and on the correlation results, by comparing the ranking provided by the metrics with the perception of Open-Source developers. In addition to the validation, in future developments we aim to extend this work by analyzing more projects, also coming from industry, and verify if the same results can be confirmed.

ARCHITECTURAL SMELLS DETECTION IN MICROSERVICES ARCHITECTURES

- *Industrial case studies on the migration towards microservices* We designed a migration to microservices process and extended Arcan to support it. We plan to extend the validation, currently limited to two case studies, on more industrial projects of larger dimension, both in the same companies or in other companies. In particular we are interested in validating the support of Arcan in the case the developer/maintainer does not have knowledge on the project under analysis. For what concerns the pos-

sible enhancements of Arcan, it would be interesting to study and develop a predictive approach based on machine learning to *predict* the candidate microservices. Predictors could take into account both static properties and semantic information of the monolithical architecture. Moreover, we plan to enhance the Arcan support for the identification of candidate microservices. A specific development that emerged from the second case study is the addition of the analysis of JSP, Javascript files (*front-end analysis*) and the SQL database to allow the user looking for relationships between the GUI and persistence layers with the Java classes.

- *Towards Microservice Smells Detection* We developed an Arcan extension and the tool AROMA, the latter based on dynamic analysis, for microservices smell detection. First of all, we aim to detect microservices smells on more and larger projects, even if it is not easy to find public repositories of this kind of projects. We aim to validate the tool also on industrial projects to collect feedback from developers and identify possible false positives. The current analysis is limited to the reconstruction of microservices and their basic dependencies, however we aim in the future to refine the identification by adding also the detection of infrastructure services, such as discovery services (e.g., Consul) and to detect other kinds of microservices smells, such as *ESB Usage* and *Shared Libraries* [236]. We are interested to include also the identification of stored service data, to enable the detection of smells regarding the usage of data (e.g., *Inappropriate Service Intimacy* [236]. Finally, we plan to integrate Arcan and AROMA with existing versioning platforms and tools, such as Github, in order to enable to exploit the two tools in continuous integration pipelines to constantly keep track of architecture erosion.

9

FINAL PERSONAL NOTE

If you reached this point, congratulations, you completed the reading of Ilaria Pigazzini's thesis. Maybe you jumped here from the start, then, I should resume in few words what I think about architectural smells and the future of this research subject. In my opinion, we only scratched the top of the big iceberg made of smells and ATD, and three PhD years are not enough for a comprehensive analysis of their causes and consequences. Given that the main goal of a student is to be objective and provide replicable, empirically measurable results, I focused on studying few but sound aspects related to AS. What I envision for future research is to start opening towards interdisciplinary studies: code is written by developers, who, even if it might sound unexpected, are real people. Real people think, decide, make mistakes, sometimes do things without a specific reason. Architectural smells are the results of human decision, but we still do not own the map of the human mind and behaviour. So, what can we do? First of all, I think future research should investigate how to retrieve as much information as possible about everyday developers process. Current approaches collect data about git commits, emails, natural languages text found in issues' trackers. It is an interesting starting point, and it poses alone many challenges: how to efficiently collect such information and how to organise them in data structures suitable for analysis. However, another interesting upgrade could be starting asking developers to be an active part of the research, by building tools for self-annotation of what happens during the development of a program. For instance, by creating a plug-in for famous IDEs able with little effort to track what's in the mind in the developer during the writing of a method.

I used the term "interdisciplinary" for a reason. Other than collecting developers data, we should also analyse the psychology of developers and architects. Why projects starts with the best intentions and end up accumulating TD? Are there external, non-code-measurable variables that we are not considering? An interesting paper from Kazman et al. [117] I read during my PhD opens with the following paragraph:

> Much empirical research in software engineering has focused on studies of "naturalistic" phenomena. But these studies collect only observational data, and so traditional analysis techniques yield only correlations between project practices and characteristics (on the one hand) and measurable outcomes (on the other hand). Without knowing

> the causal effects, it is difficult for a manager to act upon correlational evidence. For example, as source files in a software project increase in size, they tend to have more bugs and be touched by more developers. That is, file size, bugs, and number of developers are all strongly positively correlated. If one mistakes correlation for causation, then one might be tempted to conclude that bug rates could be lowered just by reducing the number of developers who are working on a file!

What it means is that we need a *causal model* able to represent the entire system surrounding the development of a software projects. Causal models are mathematical models representing causal relationships within an individual system or population, that facilitate inferences about causal relationships from statistical data [106]. Causal modeling has been largely studied by the mathematician and philosopher Judea Pearl. Pearl not only mathematically defined what a causal model is, but also developed a theory to exploit such models in practice [174]. Having such a model allows the extraction of powerful inferences and, in the case of our research field, would allow the merge of heterogeneous information, such as static code information and developers ones. For instance, Kazman in the paper proposes a causal dependency between the project's age and the number of developer, which at the same time would influence the number of bugs. However, at the moment, there are no studies aimed at delving into the development of causal models for software engineering and it is a pity, because I think it is a hard, but promising direction.

To conclude, I hope the contributions I provide with this thesis can be useful for future developments in the management of architectural smells and technical debt, and that the in next years students will dare to mix software engineering with more exotic subjects, such as causal modeling, for exciting results in this fascinating discipline.

A

APPENDIX

A.1 ADDITIONAL MATERIAL OF VALIDATION AND PERCEPTION OF THE ARCHITECTURAL SMELLS FROM THE DEVELOPERS

In this Appendix, we report the full survey and interview forms adopted in the studies described in Chapter 3.

A.1.1 *An architectural smell evaluation in an industrial context: survey questions*

The questions asked to the developers in the survey (Section 3.2) are reported in Table A.1. Each question aims to gather the developer's evaluation on specific aspects of the analyzed AS, that is particularly valuable considering their deep knowledge on the project.

The proposed questions can be grouped by category:
AS detection and awareness [Q1 – Q3, Q12]: this set of questions aim to evaluate the precision of the Arcan detection strategies and investigate the awareness of the developers on the presence of the smells.
AS impact[Q5 – Q6]: these questions aim to collect information about the perceived impact of AS on different software quality attributes.
AS refactoring[Q7 – Q9]: such questions gather information about whether refactoring activities, in the opinion of the developers, should be conducted and the type of refactoring needed to remove the smell.
AS severity, refactoring effort and priority[Q4, Q10 – Q11]: these questions aim to evaluate the effort/time needed to apply the refactoring and understand whether the smells can be ranked depending on their criticality (severity), i.e. if it is possible to quantify the smell impact thanks to the evaluation of specific smell characteristics, e.g., smell size (the number of affected classes/packages). To evaluate the answers of these questions, we define and compute three metrics (see section 5 for more details on the metrics computation), namely *Average Severity* of the smells, i.e., the average criticality that developers' associate to the smells, the *Average Effort* needed to refactor the smells and the *Average Priority of refactoring* that can be associated to the smells, i.e., the ordering of the smells depending on which should be refactored first. We chose to compute these values in order to summarize the collected data and be able to compare them.

The proposed questions are of three types: binary questions, where the possible answers are Yes or No; closed-ended questions, with multiple possible answers and open-ended questions, which were optional because we did not want to force the practitioners to spend too

much time on them and, in some cases, no answer was needed, e.g., Q3 if the smell instance is considered as a problem by the practitioner. In this way we were able to collect both quantitative and qualitative answers, in particular the latter allowed us to gain insights about the concrete opinions of the practitioners.

A.1.2 *The perception of Architectural Smells in three software companies: interview guide*

The following paragraphs report the interview guide of the study described in Section 3.3. Along with the interview guide, we prepared a one-pager containing the definition of the considered Architectural Smells (AS). The pager about AS was introduced during the interview. We first gave them the general definition of AS, then we defined the term "component", explaining that it could be referring to either a class (file) or a package (folder). Then we defined each smell, specifying whether it could be identified at only class (file) level, at only package (folder) level or both. If requested by the participant, we provided examples of AS to clarify the concept.

DEMOGRAPHICS

1. What is your current official position?
2. What is your role in your team? (day-to-day tasks example)
3. How many years of experience do you have in the current position and in total?

RQ1 QUESTIONS

1. Which of these smells are you already familiar with, from your work?
2. How many of them are there in the system you currently work on?
3. What types of smells do you think are the most important in your case? Why?

RQ2 QUESTIONS

1. What types of smells do you deem to be more detrimental for the Maintainability of the system? How are they detrimental?
2. Can you remember experiencing any issue while maintaining a class/package that could be related to an architectural smell?
3. What types of smells do you deem to be more detrimental for the Evolvability of the system? How are they detrimental?

Table A.1: Proposed questions

ID	Question	Possible Choices
Q1	Does the reported smell represent a problem in the system?	Yes or No
Q2	Were you aware of the presence of this smell in the system?	Yes or No
Q3	If it's not a problem, do you think that this could be a case of false positive AS? Or an AS not critical? For which reasons?	N/A (open-response)
Q4	How significant are the negative impacts caused by the smell in your opinion?	**0** - Not a problem **1** - Low severity **2** - Mid-Low severity **3** - Mid-High severity **4** - High severity
Q5	If it has negative impacts, which of the following software internal qualities has this type of smell an impact on?	• Reliability (R) • Efficiency (E) • Security (S) • Maintainability (M) • Other
Q6	If not removed, the impact of this type of smell get worse as time passes	**0** - Disagree **1** - Somewhat Disagree **2** - Somewhat Agree **3** - Agree
Q7	What refactoring would you suggest to conduct? (e.g. move class, extract class, extract components, extract layers, etc.. Take in consideration your best option only)	N/A (open-response)
Q8	Do you think that conducting the refactoring would create negative side-effects? If yes which ones?	N/A (open-response)
Q9	If no refactoring should be conducted, which is the reasons?	• Not a real AS (false positive) • The smell does not represent a problem because there is not a better solution • The removal of this smell is too expensive • Other
Q10	How much effort/time can be required to refactor the smell?	**0** - No refactoring needed **1** - Low ($<$ 8 h) **2** - Mid-Low (8-50 h) **3** - Mid-High (50-100 h) **4** - High ($>$100 h)
Q11	What do you think is the overall priority of refactoring this smell?	**0** - No refactoring needed **1** - Low priority **2** - Mid-Low priority **3** - Mid-High priority **4** - High priority
Q12	There is any architectural issue that you know is present in the system, but was not treated in this survey? If there is, describe it briefly	N/A (open-response)

4. In your project are there obstacles to the implementation of new features that you think are related to architectural smells? If yes, can you tell us about these obstacles?

RQ3 QUESTIONS

1. Did you try to refactor the part involved in the smell? If yes, how did you do it? If not, why?
2. Are there any practices in your team to manage the smells and their consequences?
3. Are you using any tools to monitor architectural issues (or at least dependency analysis)?
4. If yes, why did you choose that tool in particular? If not, are you aware of such tools that you would consider using?
5. What do you think is the ideal (imaginary) tool that could handle architectural smells issues?

BIBLIOGRAPHY

[1] C. van Aart, R. Pels, G. Caire, and F. Bergenti. "Creating and using ontologies in agent communication." In: *Proc. of OAS*. 2002.

[2] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension." In: *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. 2011, pp. 181–190. DOI: 10.1109/CSMR.2011.24.

[3] Hani Abdeen, Stephane Ducasse, and Houari Sahraoui. "Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software." In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 394–398. DOI: 10.1109/WCRE.2011.55.

[4] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. "Fast discovery of association rules." In: *Advances in knowledge discovery and data mining* 12.1 (1996), pp. 307–328.

[5] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules." In: *20th Int. Conf. on Very Large Data Bases*. Morgan Kaufmann, 1994, pp. 475–486.

[6] Ai Reviewer. *Ai Reviewer*. http://www.aireviewer.com/, Accessed October 2021.

[7] V. Alagarasan. "Microservices Antipatterns." In: *Microservices-Summit, NY*. 2016.

[8] Juan M Alberola, Jose M Such, Ana Garcia-Fornes, Agustin Espinosa, and Vicent Botti. "A performance evaluation of three multiagent platforms." In: *Artificial Intelligence Review* 34.2 (2010), pp. 145–176.

[9] Reem Alfayez, Pooyan Behnamghader, Kamonphop Srisopha, and Barry Boehm. "An Exploratory Study on the Influence of Developers in Technical Debt." In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 1–10. ISBN: 9781450357135. DOI: 10.1145/3194164.3194165. URL: https://doi.org/10.1145/3194164.3194165.

[10] T. Alkhaeir and B. Walter. "The Effect of Code Smells on the Relationship Between Design Patterns and Defects." In: *IEEE Access* 9 (2021), pp. 3360–3373. DOI: 10.1109/ACCESS.2020.3047870.

[11] Sara HS Almadi, Danial Hooshyar, and Rodina Binti Ahmad. "Bad Smells of Gang of Four Design Patterns: A Decade Systematic Literature Review." In: *Sustainability* 13.18 (2021), p. 10256.

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "Code2Vec: Learning Distributed Representations of Code." In: *Proc. ACM Program. Lang.* (2019). ISSN: 2475-1421.

[13] Theodoros Amanatidis, Nikolaos Mittas, Athanasia Moschou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Lefteris Angelis. "Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities." In: *Empirical Software Engineering* 25.5 (2020), pp. 4161–4204.

[14] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. "Architectural Bad Smells in Software Product Lines: An Exploratory Study." In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA '14 Companion. Sydney, Australia: ACM, 2014, 12:1–12:6. ISBN: 978-1-4503-2523-3. DOI: 10.1145/2578128.2578237. URL: http://doi.acm.org/10.1145/2578128.2578237.

[15] Apache Software Foundation. *Junit 4*. https://junit.org/junit4, Accessed October 2021.

[16] *Apache ZooKeeper*. https://zookeeper.apache.org/, Accessed October 2021.

[17] 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)*. 2019, pp. 333–337. DOI: 10.1109/SEAA.2019.00057.

[18] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita. "Automatic Metric Thresholds Derivation for Code Smell Detection." In: *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. 2015, pp. 44–53.

[19] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni. "Automatic Detection of Instability Architectural Smells." In: *Proc. of the 32nd Intern. Conf. on Software Maintenance and Evolution (ICSME 2016)*. ERA Track. Raleigh, North Carolina, USA: IEEE, Oct. 2016.

[20] F. Arcelli Fontana, R. Roveda, and M. Zanoni. "Technical Debt Indexes Provided by Tools: A Preliminary Discussion." In: *2016 IEEE 8th Inter. Work. on Managing Technical Debt (MTD)*. 2016, pp. 28–31. DOI: 10.1109/MTD.2016.11.

[21] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Andrew Tamburri, Marco Zanoni, and Elisabetta Di Nitto. "Arcan: A Tool for Architectural Smells Detection." In: *Int'l Conf. Software Architecture (ICSA 2017) Workshops*. Gothenburg, Sweden, Apr. 2017, pp. 282–285. DOI: 10.1109/ICSAW.2017.16.

[22] Francesca Arcelli Fontana and Marco Zanoni. "On investigating code smells correlations." In: *Proc. IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), RefTest Workshop*. Berlin, Germany: IEEE, 2011, pp. 474–475. DOI: 10.1109/ICSTW.2011.14.

[23] Mafruz Zaman Ashrafi, David Taniar, and Kate Smith. "Redundant Association Rules Reduction Techniques." In: *AI 2005: Advances in Artificial Intelligence*. Ed. by Shichao Zhang and Ray Jarvis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 254–263.

[24] Lerina Aversano, Umberto Carpenito, and Martina Iammarino. "An Empirical Study on the Evolution of Design Smells." In: *Information* 11.7 (2020), p. 348.

[25] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. "Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study." In: *IET software* 3.5 (2009), pp. 395–409.

[26] Paris Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Nasia Moschou, Ilaria Pigazzini, et al. "An Overview and Comparison of Technical Debt Measurement Tools." In: *IEEE Annals of the History of Computing* 01 (2020), pp. 0–0.

[27] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. "Architectural smells detected by tools: a catalogue proposal." In: *Proceedings of the Second International Conference on Technical Debt, TechDebt@ICSE 2019, Montreal, QC, Canada, May 26-27, 2019*. Ed. by Paris Avgeriou and Klaus Schmid. IEEE / ACM, 2019, pp. 88–97. ISBN: 978-1-7281-3371-3. DOI: 10.1109/TechDebt.2019.00027. URL: https://dl.acm.org/citation.cfm?id=3355348.

[28] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. "Microservices migration patterns." In: *Software: Practice and Experience* 0.0 (). DOI: 10.1002/spe.2608. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2608. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2608.

[29] Carliss Young Baldwin, Kim B Clark, Kim B Clark, et al. *Design rules: The power of modularity*. Vol. 1. MIT press, 2000.

[30] Luciano Baresi, Martin Garriga, and Alan De Renzis. "Microservices Identification Through Interface Analysis." In: *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings*. Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen. Vol. 10465. Lecture Notes in Computer Science. Springer, 2017, pp. 19–33. DOI: 10.1007/978-3-319-67262-5_2. URL: https://doi.org/10.1007/978-3-319-67262-5_2.

[31] M. S. Bartlett. "Properties of Sufficiency and Statistical Tests." In: *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* 160.901 (1937), pp. 268–282. ISSN: 00804630. URL: http://www.jstor.org/stable/96803.

[32] Alvine Boaye Belle, Ghizlane El Boussaidi, and Sègla Kpodjedo. "Combining Lexical and Structural Information to Reconstruct Software Layers." In: *Inf. Softw. Technol.* 74.C (June 2016), pp. 1–16. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.01.008. URL: http://dx.doi.org/10.1016/j.infsof.2016.01.008.

[33] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007. ISBN: 978-0-47005747-6. DOI: 10.1002/9780470058411. URL: https://doi.org/10.1002/9780470058411.

[34] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. "Pearson correlation coefficient." In: *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[35] Alexander Binun and Günter Kniesel. "DPJF - Design Pattern Detection with High Accuracy." In: *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*. 2012, pp. 245–254. DOI: 10.1109/CSMR.2012.82.

[36] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation." In: *Journal of Machine Learning Research* 3 (2003), pp. 993–1022.

[37] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann. "Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells." In: *Int. Conf. on Software Architecture Companion (ICSA-C)*. 2019, pp. 95–101.

[38] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. "Towards a Practical Maintainability Quality Model for Service- and Microservice-Based Systems." In: *Proc. of the 11th European Conference on Software Architecture*. ECSA '17. Canterbury, United Kingdom: ACM, 2017. ISBN: 9781450352178.

[39] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. USA: John Wiley & Sons, Inc., 2007. ISBN: 0470029005.

[40] L. Braubach, Winfried Lamersdorf, and A. Pokahr. "Jadex: implementing a BDI-infrastructure for JADE agents." In: *EXP In Search of Innovation (Special Issue on JADE)* 3 (Dec. 2003).

[41] S. Brin and L. Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." In: *Seventh International World-Wide Web Conference (WWW 1998)*. 1998. URL: http://ilpubs.stanford.edu:8090/361/.

[42] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. "Dynamic Itemset Counting and Implication Rules for Market Basket Data." In: *SIGMOD Rec.* 26.2 (June 1997), pp. 255–264. ISSN: 0163-5808. DOI: 10.1145/253262.253325.

[43] Daniela Briola, Viviana Mascardi, and Massimiliano Gioseffi. "OntologyBeanGenerator 5.0: Extending Ontology Concepts with Methods and Exceptions." In: *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018*. Ed. by Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Vol. 2215. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 116–123. URL: http://ceur-ws.org/Vol-2215/paper_19.pdf.

[44] Nanette Brown et al. "Managing Technical Debt in Software-Reliant Systems." In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, 47–52. ISBN: 9781450304276. DOI: 10.1145/1882362.1882373. URL: https://doi.org/10.1145/1882362.1882373.

[45] João Brunet, Roberto Almeida Bittencourt, Dalton Serey Guerrero, and Jorge C. A. de Figueiredo. "On the Evolutionary Nature of Architectural Violations." In: *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. 2012, pp. 257–266. DOI: 10.1109/WCRE.2012.35.

[46] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. "From Monolithic to Microservices: An Experience Report from the Banking Domain." In: *IEEE Software* 35.03 (2018), pp. 50–55. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141026.

[47] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. "Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations." In: New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450380379. DOI: 10.1145/3404835.3462840. URL: https://doi.org/10.1145/3404835.3462840.

[48] Yuanfang Cai and Rick Kazman. "DV8: automated architecture analysis tool suites." In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2019, pp. 53–54.

[49] Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. "Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture." In: *IEEE Transactions on Software Engineering* 45.7 (2019), pp. 657–682. DOI: 10.1109/TSE.2018.2797899.

[50] Rafael Capilla, Tommi Mikkonen, Carlos Carrillo, Francesca Arcelli Fontana, Ilaria Pigazzini, and Valentina Lenarduzzi. "Impact of Opportunistic Reuse Practices to Technical Debt." In: *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2021, pp. 16–25. DOI: 10.1109/TechDebt52882.2021.00011.

[51] Rafael C. Cardoso, Angelo Ferrando, Daniela Briola, Claudio Menghi, and Tobias Ahlbrecht. "Agents and Robots for Reliable Engineered Autonomy: A Perspective from the Organisers of AREA 2020." In: *J. Sens. Actuator Networks* 10.2 (2021), p. 33. DOI: 10.3390/jsan10020033. URL: https://doi.org/10.3390/jsan10020033.

[52] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. "Migrating towards Microservices: Migration and Architecture Smells." In: *Proc. IWoR*. ACM, 2018, pp. 1–6. DOI: 10.1145/3242163.3242164.

[53] S. R. Chidamber and C. F. Kemerer. "A Metric Suite for Object-Oriented Design." In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 293–318.

[54] H. Christopher Frey and Sumeet R. Patil. "Identification and Review of Sensitivity Analysis Methods." In: *Risk Analysis* 22.3 (2002), pp. 553–578.

[55] *ClassCycle*. http://classycle.sourceforge.net/, Accessed October 2021.

[56] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[57] *Consul*. https://www.consul.io/, Accessed October 2021.

[58] Anna Corazza, Sergio Martino, Valerio Maggio, and Giuseppe Scanniello. "Weighing Lexical Information for Software Clustering in the Context of Architecture Recovery." In: *Empirical Softw. Engg.* 21.1 (Feb. 2016), pp. 72–103. ISSN: 1382-3256.

[59] F. Corno, L. De Russis, and J. P. Sáenz. "How is Open Source Software Development Different in Popular IoT Projects?" In: *IEEE Access* 8 (2020), pp. 28337–28348. DOI: 10.1109/ACCESS.2020.2972364.

[60] Oracle Corporation. *The Java EE 5 Tutorial*. URL: https://docs.oracle.com/javaee/5/tutorial/doc/index.html. (Accessed October 2021).

[61] Harald Cramér. *Mathematical methods of statistics*. Vol. 43. Princeton university press, 1999.

[62] Ward Cunningham. "The WyCash portfolio management system." In: *OOPS Messenger* 4.2 (1993), pp. 29–30. DOI: 10.1145/157710.157715.

[63] *Dependency Finder*. http://depfind.sourceforge.net/, Accessed October 2021.

[64] Designite. *Tushar Sharma, Pratibha Mishra and Rohit Tiwari*. Accessed October 2021. URL: www.designite-tools.com.

[65] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. "Migrating towards Microservice Architectures: an Industrial Survey." In: *IEEE International Conference on Software Architecture (ICSA 2018)*. Seattle, USA: IEEE, 2018.

[66] J.A. Díaz-Pace, A. Tommasel, and D. Godoy. "Towards Anticipation of Architectural Smells Using Link Prediction Techniques." In: *18th IEEE SCAM, 2018*.

[67] George Digkas, Alexander N Chatzigeorgiou, Apostolos Ampatzoglou, and Paris C Avgeriou. "Can Clean New Code reduce Technical Debt Density." In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/TSE.2020.3032557.

[68] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. "The evolution of technical debt in the apache ecosystem." In: *European Conference on Software Architecture*. Springer. 2017, pp. 51–66.

[69] Regine Endsuleit, Jacques Calmet, et al. "A Security Analysis on JADE (-S) V. 3.2." In: *Proceedings of NORDSEC*. Citeseer. 2005, pp. 20–28.

[70] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. *Evaluation of Microservice Architectures: A Metric and Tool-Based Approach*. Ed. by Jan Mendling and Haralambos Mouratidis. Cham: Springer, 2018, pp. 74–89. ISBN: 978-3-319-92901-9.

[71] Steven D Eppinger and Tyson R Browning. *Design structure matrix methods and applications*. MIT press, 2012.

[72] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt." In: *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Italy: ACM, 2015, pp. 50–60. ISBN: 978-1-4503-3675-8.

[73] Yong Fang, Cheng Huang, Yu Su, and Yaoyao Qiu. "Detecting malicious JavaScript code based on semantic analysis." In: *Computers Security* 93 (2020), p. 101764. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.101764. URL: https://www.sciencedirect.com/science/article/pii/S0167404820300481.

[74] *Feign*. https://github.com/OpenFeign/feign, Accessed October 2021.

[75] Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, Alexander Chatzigeorgiou, and Elisa.Y. Nakagawa. "What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes?" In: *Information and Software Technology* 105 (2019), pp. 1 –16. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.07.014.

[76] Daniel Feitosa, Paris Avgeriou, Apostolos Ampatzoglou, and Elisa Yumi Nakagawa. "The evolution of design pattern grime: an industrial case study." In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2017, pp. 165–181.

[77] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. "Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification." In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 609–613. DOI: 10.1109/SANER.2016.84.

[78] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. "Are architectural smells independent from code smells? An empirical study." In: *Journal of Systems and Software* 154 (2019), pp. 139 –156. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.04.066. URL: http://www.sciencedirect.com/science/article/pii/S0164121219301013.

[79] Francesca Arcelli Fontana, Federico Locatelli, Ilaria Pigazzini, and Paolo Mereghetti. "An architectural smell evaluation in an industrial context." In: *The Fifteenth International Conference on Software Engineering Advances, ICSEA 2020, 18-22 October 2020, Porto, Portugal*. 2020.

[80] Francesca Arcelli Fontana and Ilaria Pigazzini. "Evaluating the Architectural Debt of IoT Projects." In: *3rd IEEE/ACM International Workshop on Software Engineering Research and Practices for the IoT SERP4IoT 2021, Madrid, Spain, June 3, 2021*. IEEE, 2021, pp. 27–31. DOI: 10.1109/SERP4IoT52556.2021.00011. URL: https://doi.org/10.1109/SERP4IoT52556.2021.00011.

[81] Francesca Arcelli Fontana and Ilaria Pigazzini. "Evaluating the Architectural Debt of IoT Projects." In: *3rd IEEE/ACM International Workshop on Software Engineering Research and Practices for the IoT SERP4IoT 2021, Madrid, Spain, June 3, 2021*. IEEE, 2021, pp. 27–31. DOI: 10.1109/SERP4IoT52556.2021.00011. URL: https://doi.org/10.1109/SERP4IoT52556.2021.00011.

[82] Francesca Arcelli Fontana, Ilaria Pigazzini, Claudia Raibulet, Stefano Basciano, and Riccardo Roveda. "PageRank and criticality of architectural smells." In: *Proceedings of the 13th European Conference on Software Architecture, ECSA 2019, Paris, France, September 9-13, 2019, Companion Proceedings (Proceedings Volume 2)*, 2019, pp. 197–204. DOI: 10.1145/3344948.3344982. URL: https://doi.org/10.1145/3344948.3344982.

[83] Francesca Arcelli Fontana and Marco Zanoni. "Code smell severity classification using machine learning techniques." In: *Knowl. Based Syst.* 128 (2017).

[84] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420. URL: http://portal.acm.org/citation.cfm?id=579257.

[85] P. D. Francesco, I. Malavolta, and P. Lago. "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption." In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 21–30. DOI: 10.1109/ICSA.2017.24.

[86] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. "From Monolith to Microservices: A Classification of Refactoring Approaches." In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer International Publishing, 2019, pp. 128–141.

[87] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency." In: *IEEE Software* 35.3 (2018), pp. 63–72. ISSN: 0740-7459.

[88] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[89] S.G. Ganesh, Tushar Sharma, and Girish Suryanarayana. "Towards a Principle-based Classification of Structural Design Smells." In: *Journal of Object Technology* 12.2 (June 2013), 1:1–29. ISSN: 1660-1769. DOI: 10.5381/jot.2013.12.2.a1.

[90] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Identifying Architectural Bad Smells." In: *CSMR 2009*. Germany: IEEE, 2009, pp. 255–258. DOI: 10.1109/CSMR.2009.59.

[91] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Toward a Catalogue of Architectural Bad Smells." In: *Proceedings of the 5th International Conference on the Quality of Software Architectures (QoSA 2009)*. East Stroudsburg, PA, USA: Springer Berlin Heidelberg, June 2009, pp. 146–162. ISBN: 978-3-642-02351-4. DOI: 10.1007/978-3-642-02351-4_10.

[92] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. "Enhancing Architectural Recovery Using Concerns." In: *Proc. of ASE 2011*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 552–555. ISBN: 978-1-4577-1638-6.

[93] GitHub, Inc. *Github*. https://github.com/, Accessed October 2021.

[94] Google. *Snapshot API Docs*. https://guava.dev/releases/snapshot-jre/api/docs/,

[95] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems." In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017.

[96] Giona Granchelli, Mario Cardarelli, Paolo Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. "Towards Recovering the Software Architecture of Microservice-Based Systems." In: Apr. 2017, pp. 46–53.

[97] G. Grano, F. Palomba, and H. C. Gall. "Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators." In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[98] T. L. Griffiths and M. Steyvers. "Finding Scientific Topics." In: *Proceedings of the National Academy of Sciences* 101.Suppl. 1 (2004), pp. 5228–5235.

[99] Thirupathi Guggulothu and Salman Abdul Moiz. "An approach to suggest code smell order for refactoring." In: *International Conference on Emerging Technologies in Computer Engineering*. Springer. 2019, pp. 250–260.

[100] Jilles van Gurp and Jan Bosch. "Design erosion: problems and causes." In: *Journal of Systems and Software* 61.2 (2002), pp. 105–119. DOI: 10.1016/S0164-1212(01)00152-2.

[101] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. "Service Cutter: A Systematic Approach to Service Decomposition." In: *5th European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Ed. by Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski. Vol. LNCS-9846. Service-Oriented and Cloud Computing. Part 5: Compositionality. Vienna, Austria: Springer International Publishing, Sept. 2016, pp. 185–200. DOI: 10.1007/978-3-319-44482-6_12. URL: https://hal.inria.fr/hal-01638590.

[102] Thomas Haitzer, Elena Navarro, and Uwe Zdun. "Reconciling software architecture and source code in support of software evolution." In: *Journal of Systems and Software* 123 (2017), pp. 119–144.

[103] Headway Software Technologies Ltd. *Stucture101*. https://structure101.com/, Accessed October 2021.

[104] Vincent J. Hellendoorn and Premkumar Devanbu. "Are Deep Neural Networks the Best Choice for Modeling Source Code?" In: *Proc. of ESEC/FSE 2017*. Paderborn, Germany, 2017. ISBN: 978-1-4503-5105-8.

[105] Sebastian Herold. "An Initial Study on the Association Between Architectural Smells and Degradation." In: *Software Architecture*. Ed. by Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann. Cham: Springer International Publishing, 2020, pp. 193–201. ISBN: 978-3-030-58923-3.

[106] Christopher Hitchcock. "Causal Models." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2020. Metaphysics Research Lab, Stanford University, 2020.

[107] Sunny Huynh, Yuanfang Cai, and Kanwarpreet Sethi. "Design rule hierarchy and analytical decision model transformation." In: *Drexel University, Philadelphia, PA, USA, Tech. Rep. DU-CS-08-04* (2008).

[108] ISO/IEC 25010. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011.

[109] Clemente Izurieta and James M. Bieman. "A multiple case study of design pattern decay, grime, and rot in evolving software systems." In: *Software Quality Journal* 21.2 (2013), pp. 289–323. ISSN: 1573-1367. DOI: 10.1007/s11219-012-9175-x.

[110] *JArchitect*. https://www.jarchitect.com/, Accessed October 2021.

[111] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. "Analysing Anti-patterns Static Relationships with Design Patterns." In: *Electronic Communications of the European Association for the Study of Science and Technology* 59 (2013). DOI: 10.14279/tuj.eceasst.59.930.

[112] Jagadeesh Jagarlamudi, Hal Daumé III, and Raghavendra Udupa. "Incorporating Lexical Priors into Topic Models." In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. EACL '12. Avignon, France: Association for Computational Linguistics, 2012, pp. 204–213. ISBN: 978-1-937284-19-0. URL: http://dl.acm.org/citation.cfm?id=2380816.2380844.

[113] *Jira*. https://www.atlassian.com/software/jira, Accessed October 2021.

[114] KNIME AG. *KNIME Analytics Platform*. https://www.knime.com/knime-analytics-platform, Accessed October 2021.

[115] Henry F Kaiser. "A second generation little jiffy." In: *Psychometrika* 35.4 (1970), pp. 401–415.

[116] R. Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, S. Haziyev, V. Fedak, and A. Shapochka. "A Case Study in Locating the Architectural Roots of Technical Debt." In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE Int. Conf. on*. Vol. 2. 2015, pp. 179–188. DOI: 10.1109/ICSE.2015.146.

[117] Rick Kazman, Robert Stoddard, David Danks, and Yuanfang Cai. "Causal Modeling, Discovery, amp; Inference for Software Engineering." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 172–174. DOI: 10.1109/ICSE-C.2017.138.

[118] G. Kecskemeti, A. C. Marosi, and A. Kertesz. "The ENTICE approach to decompose monolithic services into microservices." In: *2016 International Conference on High Performance Computing Simulation (HPCS)*. 2016, pp. 591–596.

[119] M.G. Kendall and J.D. Gibbons. *Rank Correlation Methods*. Charles Griffin Book. E. Arnold, 1990. ISBN: 9780852643051.

[120] Foutse Khomh and Yann-Gaël Guéhéneuc. "Do Design Patterns Impact Software Quality Positively?" In: *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*. 2008, pp. 274–278. DOI: 10.1109/CSMR.2008.4493325.

[121] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns." In: *Journal of Systems and Software* 84.4 (2011). The Ninth International Conference on

Quality Software, pp. 559–572. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.11.921.

[122] Martin Kleehaus, Ömer Uludağ, Patrick Schäfer, and Florian Matthes. "MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments." In: *Int. Conf. on Advanced Information Systems Engineering*. Springer. 2018, pp. 148–162.

[123] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. "A study on the role of software architecture in the evolution and quality of software." In: vol. 2015-Augus. IEEE International Working Conference on Mining Software Repositories. IEEE Computer Society, 2015, pp. 246–257. ISBN: 9780769555942. DOI: 10.1109/MSR.2015.30.

[124] Philippe Kruchten. "An ontology of architectural design decisions in software intensive systems." In: *2nd Groningen workshop on software variability*. Citeseer. 2004, pp. 54–61.

[125] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. "Technical Debt: From Metaphor to Theory and Practice." In: *IEEE Softw.* 29.6 (2012), pp. 18–21.

[126] *Kubernetes*. . https://www.ibm.com/it-it/cloud/kubernetes-service/, Accessed October 2021.

[127] Michele Lanza. "The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques." In: *4th IWPSE*. Vienna, Austria: ACM, 2001, pp. 37–42. ISBN: 1-58113-508-4.

[128] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[129] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja. "Software Engineering for the Internet of Things." In: *IEEE Software* 34.1 (2017), pp. 24–28. DOI: 10.1109/MS.2017.28.

[130] *Lattix*. http://lattix.com/, Accessed October 2021.

[131] Jannik Laval and Stéphane Ducasse. "Resolving cyclic dependencies between packages with Enriched Dependency Structural Matrix." In: *Software: Practice and Experience* (Nov. 2012). URL: https://hal.inria.fr/hal-00748120.

[132] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. "An Empirical Study of Architectural Change in Open-Source Software Systems." In: *Working Conference on Mining Software Repositories*. 2015, pp. 235–245.

[133] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. "An Empirical Study of Architectural Decay in Open-Source Software." In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 176–17609. DOI: 10.1109/ICSA.2018.00027.

[134] Kwanwoo Lee, Kyo C. Kang, Wonsuk Chae, and Byoung Wook Choi. "Feature-based approach to object-oriented engineering of applications for reuse." In: *Software: Practice and Experience* 30.9 (2000), pp. 1025–1046. DOI: 10.1002/1097-024X(20000725).

[135] Jason Lefever, Yuanfang Cai, Humberto Cervantes, Rick Kazman, and Hongzhou Fang. "On the Lack of Consensus Among Technical Debt Detection Tools." In: *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 121–130. DOI: 10.1109/ICSE-SEIP52600.2021.00021. URL: https://doi.org/10.1109/ICSE-SEIP52600.2021.00021.

[136] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi. "Are SonarQube Rules Inducing Bugs?" In: *27th International Conference on Software Analysis, Evolution and Reengineering (SANER2020)*. 2020, pp. 501–511.

[137] Valentina Lenarduzzi, Teemu Orava, Nyyti Saarimäki, Kari Systa, and Davide Taibi. "An Empirical Study on Technical Debt in a Finnish SME." In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870169.

[138] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. "Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study." In: *Journal of Systems and Software* 170 (2020), p. 110750.

[139] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. "Analyzing Forty Years of Software Maintenance Models." In: *39th International Conference on Software Engineering Companion*. ICSE-C '17. 2017, pp. 146–148.

[140] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. "A Survey on Code Analysis Tools for Software Maintenance Prediction." In: *6th International Conference in Software Engineering for Defence Applications*. Springer International Publishing, 2020, pp. 165–175.

[141] Zengyang Li, Peng Liang, and Paris Avgeriou. "Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios." In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*. 2015, pp. 65–74. DOI: 10.1109/WICSA.2015.19.

[142] Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi, and Apostolos Ampatzoglou. "An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt." In: *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '14. Marcq-en-Bareul, France: Association for Computing Machinery, 2014, 119–128. ISBN: 9781450325769. DOI: 10.1145/2602576.2602581. URL: https://doi.org/10.1145/2602576.2602581.

[143] E. Linstead, C. Lopes, and P. Baldi. "An Application of Latent Dirichlet Allocation to Analyzing Software Evolution." In: *2008 Seventh International Conference on Machine Learning and Applications*. IEEE, 2008, pp. 813–818. DOI: 10.1109/ICMLA.2008.47.

[144] Martin Lippert and Stephen Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, Apr. 2006, p. 286. ISBN: 978-0-470-85892-9.

[145] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping." In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[146] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. "Power Laws in Software." In: *ACM Trans. Softw. Eng. Methodol.* 18.1 (Oct. 2008). ISSN: 1049-331X. DOI: 10.1145/1391984.1391986. URL: https://doi.org/10.1145/1391984.1391986.

[147] N. Lu, G. Glatz, and D. Peuser. "Moving mountains – practical approaches for moving monolithic applications to Microservices." In: *International Conference on Microservices (Microservices 2019)*. Dortmund, Germany, 2019.

[148] Radu Marinescu. "Assessing technical debt by identifying design flaws in software systems." In: *IBM Journal of Research and Development* 56.5 (2012), 9:1–9:13. ISSN: 0018-8646. DOI: 10.1147/JRD.2012.2204512.

[149] G. Marquez and H. Astudillo. "Actual Use of Architectural Patterns in Microservices-based Open Source Projects." In: *Asia-Pacific Software Engineering Conference (APSEC 2018)*. 2018.

[150] Robert C. Martin. "Object Oriented Design Quality Metrics: An Analysis of dependencies." In: *ROAD* 2.3 (1995).

[151] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2007.

[152] Antonio Martini and Jan Bosch. "An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt." In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 31–40.

[153] Antonio Martini, Jan Bosch, and Michel Chaudron. "Architecture Technical Debt: Understanding Causes and a Qualitative Model." In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2014, pp. 85–92. DOI: 10.1109/SEAA.2014.65.

[154] Antonio Martini, Jan Bosch, and Michel Chaudron. "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study." In: *Information and Software Technology* 67 (2015), pp. 237–253. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.07.005. URL: https://www.sciencedirect.com/science/article/pii/S0950584915001287.

[155] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. "Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company." In: *Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings*. 2018, pp. 320–335. DOI: 10.1007/978-3-030-00761-4_21.

[156] Antonio Martini, Erik Sikander, and Niel Madlani. "A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component." In: *Information and Software Technology* 93 (2018), pp. 264–279. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.08.005. URL: https://www.sciencedirect.com/science/article/pii/S095058491630355X.

[157] Viviana Mascardi, Danny Weyns, and Alessandro Ricci. "Engineering Multi-Agent Systems: State of Affairs and the Road Ahead." In: *ACM SIGSOFT Softw. Eng. Notes* 44.1 (2019), pp. 18–28. DOI: 10.1145/3310013.3310035. URL: https://doi.org/10.1145/3310013.3310035.

[158] *Massey Archtiecture Explorer*. Not available, unreachable website.

[159] B. Mayer and R. Weinreich. "An Approach to Extract the Architecture of Microservice-Based Software Systems." In: *(SOSE)*. 2018, pp. 21–30.

[160] Genc Mazlami, Jürgen Cito, and Philipp Leitner. "Extraction of Microservices from Monolithic Software Architectures." In: *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*. 2017.

[161] Camilo Mendoza, Kelly Garcés, Rubby Casallas, and José Bocanegra. "Detecting Architectural Issues During the Continuous Integration Pipeline." In: *2019 ACM/IEEE 22nd Interna-*

tional Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). 2019, pp. 589–597. DOI: 10.1109/MODELS-C.2019.00090.

[162] Mayank Mishra, Shruti Kunde, and Manoj Nambiar. "Cracking the Monolith: Challenges in Data Transitioning to Cloud Native Architectures." In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA '18. Madrid, Spain: ACM, 2018, 35:1–35:4. ISBN: 978-1-4503-6483-6. DOI: 10.1145/3241403.3241440. URL: http://doi.acm.org/10.1145/3241403.3241440.

[163] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells." In: *12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, Montreal, QC, Canada, May 4-8, 2015*. 2015, pp. 51–60. DOI: 10.1109/WICSA.2015.12.

[164] N. Moha, Y. Gueheneuc, and P. Leduc. "Automatic Generation of Detection Algorithms for Design Defects." In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, pp. 297–300.

[165] Naouel Moha, Duc-loc Huynh, Yann-Gaël Guéhéneuc, and Ptidej Team. "A taxonomy and a first study of design pattern defects." In: *IEEE International Workshop on Software Technology and Engineering Practice (STEP) 2005* (2005), p. 225.

[166] *Mongo DB*. https://www.mongodb.com, Accessed October 2021.

[167] ABM Moniruzzaman and Syed Akhter Hossain. "NoSQL Database: New Era of Databases for Big data Analytics-Classification, Characteristics and Comparison." In: *International Journal of Database Theory and Application* 6.4 (), pp. 1–13.

[168] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues. "The squale model — A practice-based industrial quality model." In: *2009 IEEE International Conference on Software Maintenance*. 2009, pp. 531–534.

[169] Luis Mulet, Jose M Such, and Juan M Alberola. "Performance evaluation of open-source multiagent platforms." In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. 2006, pp. 1107–1109.

[170] *NDepend*. https://www.ndepend.com/, Accessed October 2021.

[171] Nachiappan Nagappan and Thomas Ball. "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study." In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). 2007, pp. 364–373.

[172] Lloyd S Nelson. "The Anderson-Darling test for normality." In: *Journal of Quality Technology* 30.3 (1998), p. 298.

[173] *Neo4j*. https://neo4j.com/, Accessed October 2021.

[174] Leland Gerson Neuberg. "Causality: models, reasoning, and inference, by judea pearl, cambridge university press, 2000." In: *Econometric Theory* 19.4 (2003), pp. 675–685.

[175] Sam Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357.

[176] T. H. Ng, Yuen Tak Yu, S. C. Cheung, and W. K. Chan. "Human and Program Factors Affecting the Maintenance of Programs with Deployed Design Patterns." In: *Inf. Softw. Technol.* 54.1 (Jan. 2012), pp. 99–118. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.08.002.

[177] Robert L Nord, Ipek Ozkaya, Philippe Kruchten, and Marco Gonzalez-Rojas. "In search of a metric for managing architectural technical debt." In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 91–100.

[178] Robert L. Nord, Ipek Ozkaya, Edward J. Schwartz, Forrest Shull, and Rick Kazman. "Can Knowledge of Technical Debt Help Identify Software Vulnerabilities?" In: *9th Workshop on Cyber Security Experimentation and Test, CSET '16, Austin, TX, USA, August 8, 2016*. Ed. by Eric Eide and Mathias Payer. USENIX Association, 2016. URL: https://www.usenix.org/conference/cset16/workshop-program/presentation/nord.

[179] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. "An Empirical Model of Technical Debt and Interest." In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. MTD '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, 1–8. ISBN: 9781450305860. DOI: 10.1145/1985362.1985364. URL: https://doi.org/10.1145/1985362.1985364.

[180] Odysseus Software GmbH. *STAN*. http://stan4j.com/. Accessed 2021.

[181] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. "The evolution and impact of code smells: A case study of two open source systems." In: *2009 3rd international symposium on empirical software engineering and measurement*. IEEE. 2009, pp. 390–400.

[182] Anderson Oliveira, Leonardo Sousa, Willian Oizumi, and Alessandro Garcia. "On the Prioritization of Design-Relevant Smelly Elements: A Mixed-Method, Multi-Project Study." In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '19. Salvador, Brazil: Association for Computing Machinery, 2019, 83–92. ISBN: 9781450376372.

[183] Paul Oman and Jack Hagemeister. "Construction and testing of polynomials predicting software maintainability." In: *Journal of Systems and Software* 24.3 (1994). Oregon Workshop on Software Metrics, pp. 251–266. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(94)90067-1. URL: https://www.sciencedirect.com/science/article/pii/0164121294900671.

[184] *OpenZipkin a Java 1.8+ service, packaged as an executable jar.* URL: https://github.com/openzipkin/zipkin/tree/master/zipkin-server.

[185] T. D. Oyetoyan, J. Falleri, J. Dietrich, and K. Jezek. "Circular dependencies and change-proneness: An empirical study." In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 241–250.

[186] Ipek Ozkaya. "Can We Really Achieve Software Quality?" In: *IEEE Software* 38.3 (2021), pp. 3–6. DOI: 10.1109/MS.2021.3060552.

[187] Wei-feng PAN, Bing LI, Yu-tao MA, and Bo JIANG. "Identifying the key packages using weighted PageRank algorithm." In: *ACTA ELECTONICA SINICA* 42.11 (2014), p. 2174.

[188] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. "Mining Version Histories for Detecting Code Smells." In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 462–489.

[189] Pankesh Patel and Damien Cassou. "Enabling high-level application development for the Internet of Things." In: *Journal of Systems and Software* 103 (2015), pp. 62 –84. ISSN: 0164-1212.

[190] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. "Developer-Driven Code Smell Prioritization." In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 220–231. ISBN: 9781450375177.

[191] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. "Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching." In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, pp. 578–589.

[192] Fabrizio Perin, Lukas Renggli, and Jorge Ressia. "Ranking software artifacts." In: *4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010)*. Vol. 120. Citeseer. 2010.

[193] Ralph Peters and Andy Zaidman. "Evaluating the lifespan of code smells using software repository mining." In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE. 2012, pp. 411–416.

[194] Bżej Pietrzak and Bartosz Walter. "Leveraging Code Smell Detection with Inter-smell Relations." In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi. Vol. 4044. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 75–84. ISBN: 978-3-540-35094-1. DOI: 10.1007/11774129_8.

[195] Ilaria Pigazzini. "Automatic detection of architectural bad smells through semantic representation of code." In: *Proceedings of the 13th European Conference on Software Architecture, ECSA 2019, Paris, France, September 9-13, 2019, Companion Proceedings (Proceedings Volume 2)*, ed. by Laurence Duchien, Anne Koziolek, Raffaela Mirandola, Elena Maria Navarro Martínez, Clément Quinton, Riccardo Scandariato, Patrizia Scandurra, Catia Trubiani, and Danny Weyns. ACM, 2019, pp. 59–62. DOI: 10.1145/3344948.3344951. URL: https://doi.org/10.1145/3344948.3344951.

[196] Ilaria Pigazzini, Daniela Briola, and Francesca Arcelli Fontana. "Architectural Technical Debt of Multi-Agent Systems Development Platforms." In: *Proceedings of the 22nd Workshop "From Objects to Agents", Bologna, Italy, September 1-3, 2021*. Ed. by Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor. Vol. 2963. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 1–13. URL: http://ceur-ws.org/Vol-2963/./paper13.pdf.

[197] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. *Towards Microservice Smells Detection*. TechDebt '20. Seoul, Republic of Korea: ACM, 2020, 92–97. ISBN: 9781450379601.

[198] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. "Tool Support for the Migration to Microservice Architecture: An Industrial Case Study." In: *Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings*. 2019, pp. 247–263. DOI: 10.1007/978-3-030-29983-5_17. URL: https://doi.org/10.1007/978-3-030-29983-5_17.

[199] Ilaria Pigazzini, Francesca Arcelli Fontana, and Bartosz Walter. "A study on correlations between architectural smells and design patterns." In: *J. Syst. Softw.* 178 (2021), p. 110984. DOI: 10.1016/j.jss.2021.110984. URL: https://doi.org/10.1016/j.jss.2021.110984.

[200] Ilaria Pigazzini, Davide Foppiani, and Francesca Arcelli Fontana. "Two Different Facets of Architectural Smells Criticality: An Empirical Study." In: *ECSA 2021 Companion Volume, Virtual (originally: Växjö, Sweden), 13-17 September, 2021*. Ed. by Robert

Heinrich, Raffaela Mirandola, and Danny Weyns. Vol. 2978. CEUR Workshop Proceedings. CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-2978/msr4sa-paper2.pdf.

[201] Aniket Potdar and Emad Shihab. "An exploratory study on self-admitted technical debt." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 91–100.

[202] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta. "A controlled experiment in maintenance: comparing design patterns to simpler solutions." In: *IEEE Transactions on Software Engineering* 27.12 (2001), pp. 1134–1144. ISSN: 2326-3881. DOI: 10.1109/32.988711.

[203] Girish Maskeri Rama, Santonu Sarkar, and Kenneth Heafield. "Mining business topics in source code using latent dirichlet allocation." In: *Proceeding of the 1st Annual India Software Engineering Conference, ISEC 2008, Hyderabad, India, February 19-22, 2008*. Ed. by Gautam Shroff, Pankaj Jalote, and Sriram K. Rajamani. ACM, 2008, pp. 113–120. ISBN: 978-1-59593-917-3.

[204] Juan Ramos et al. "Using tf-idf to determine word relevance in document queries." In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. Piscataway, NJ. 2003, pp. 133–142.

[205] A. Rani and J. K. Chhabra. "Prioritization of smelly classes: A two phase approach (Reducing refactoring efforts)." In: *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*. 2017.

[206] Leevi Rantala and Mika Mäntylä. "Predicting technical debt from commit contents: reproduction and extension with automated feature selection." In: *Software Quality Journal* 28.4 (2020), pp. 1551–1579.

[207] M. Richards. "Microservices AntiPatterns and Pitfalls." In: *O Reilly eBooks* (2016).

[208] C. Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications Company, 2018. ISBN: 9781617294549. URL: https://books.google.it/books?id=UeK1swEACAAJ.

[209] Luca Rizzi, Francesca Arcelli Fontana, and Riccardo Roveda. "Support for architectural smell refactoring." In: *2nd Int. Workshop on Refactoring, IWoR@ASE 2018, Montpellier, France, September 4, 2018*. 2018, pp. 7–10. DOI: 10.1145/3242163.3242165. URL: https://doi.org/10.1145/3242163.3242165.

[210] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni. "Towards an Architectural Debt Index." In: *44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018, Prague, Czech Republic, August 29-31, 2018*. 2018, pp. 408–416.

[211] Per Runeson and Martin Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering." In: *Empirical Softw. Engg.* 14.2 (Apr. 2009), pp. 131–164. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9102-8. URL: http://dx.doi.org/10.1007/s10664-008-9102-8.

[212] Nafiseh Sadat Jalali, Habib Izadkhah, and Shahriar Lotfi. "Multi-objective search-based software modularization: structural and non-structural features." In: *Soft Computing* (Nov. 2018). DOI: 10.1007/s00500-018-3666-z.

[213] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Context-based approach to prioritize code smells for refactoring." In: *Journal of Software: Evolution and Process* (2017), e1886.

[214] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. New York, NY, USA: Halsted Press, 2004. ISBN: 0470870931.

[215] Jorge Andrés Díaz Pace Santiago A. Vidal Claudia Marcos. "An approach to prioritize code smells for refactoring." In: *Autom. Softw. Eng.* 23.3 (2016), pp. 501–532.

[216] Darius Sas, Paris Avgeriou, and Francesca Arcelli Fontana. "Investigating Instability Architectural Smells Evolution: An Exploratory Case Study." In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 557–567. ISBN: 978-1-7281-3094-1. DOI: 10.1109/ICSME.2019.00090. URL: https://doi.org/10.1109/ICSME.2019.00090.

[217] Darius Sas, Ilaria Pigazzini, Paris Avgeriou, and Francesca Arcelli Fontana. "The perception of Architectural Smells in industrial practice." In: *IEEE Software* (2021), pp. 0–0. DOI: 10.1109/MS.2021.3103664.

[218] Robert Sedgewick and Kevin Wayne. *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016. ISBN: 978-0-1343-8468-9.

[219] Samuel Sanford Shapiro and Martin B Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611.

[220] Tushar Sharma. "How Deep is the Mud: Fathoming Architecture Technical Debt Using Designite." In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019, pp. 59–60. DOI: 10.1109/TechDebt.2019.00018.

[221] Tushar Sharma, Paramvir Singh, and Diomidis Spinellis. "An empirical investigation on the relationship between design and architecture smells." In: *Empirical Software Engineering* (2020), pp. 1–49.

[222] Miltiadis Siavvas, Dimitrios Tsoukalas, Marija Jankovic, Dionysios Kehagias, and Dimitrios Tzovaras. "Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises." In: *Enterprise Information Systems* (Sept. 2020). DOI: 10.1080/17517575.2020.1824017.

[223] Lakshitha de Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey." In: *Journal of Systems and Software* 85.1 (2012). DOI: 10.1016/j.jss.2011.07.036.

[224] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. "The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures." In: *Software: Practice and Experience* (2021).

[225] SonarSource S.A. *SonarQube*. http://www.sonarqube.org/. 2015.

[226] Bruno L Sousa, Mariza AS Bigonha, and Kecia AM Ferreira. "An exploratory study on cooccurrence of design patterns and bad smells using software metrics." In: *Software: Practice and Experience* 49.7 (2019), pp. 1079–1113.

[227] C. Spearman. "The Proof and Measurement of Association between Two Things." In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101. ISSN: 00029556. URL: http://www.jstor.org/stable/1412159.

[228] Diomidis Spinellis. "How to Select Open Source Components." In: *Computer* 52.12 (2019), pp. 103–106.

[229] *Spring framework*. https://spring.io/, Accessed October 2021.

[230] Peter Strečanský, Stanislav Chren, and Bruno Rossi. "Comparing Maintainability Index, SIG Method, and SQALE for Technical Debt Identification." In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic: Association for Computing Machinery, 2020, 121–124. ISBN: 9781450368667. DOI: 10.1145/3341105.3374079. URL: https://doi.org/10.1145/3341105.3374079.

[231] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0128013974, 9780128013977.

[232] D. Taibi, V. Lenarduzzi, and C. Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation." In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.

[233] D. Taibi, V. Lenarduzzi, and C. Pahl. "Architectural Patterns for Microservices: a Systematic Mapping Study." In: *Int. Conf. on Cloud Computing and Services Science (CLOSER2018)* (2018).

[234] D. Taibi, V. Lenarduzzi, and C. Pahl. "Microservices Anti-patterns: A Taxonomy." In: *Microservices*. Springer International Publishing, Dec. 2019, pp. 111–128. DOI: 10.1007/978-3-030-31646-4_5. URL: https://doi.org/10.1007/978-3-030-31646-4_5.

[235] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. "How developers perceive smells in source code: A replicated study." In: *Information and Software Technology* 92.Supplement C (2017), pp. 223 –235. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0950584916304128.

[236] Davide Taibi and Valentina Lenarduzzi. "On the Definition of Microservice Bad Smells." In: *IEEE Software* 35.3 (2018), pp. 56–62. DOI: 10.1109/MS.2018.2141031. URL: https://doi.org/10.1109/MS.2018.2141031.

[237] Jie Tan, Daniel Feitosa, and Paris Avgeriou. "An Empirical Study on Self-Fixed Technical Debt." In: *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 11–20. ISBN: 9781450379601. DOI: 10.1145/3387906.3388621. URL: https://doi.org/10.1145/3387906.3388621.

[238] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies." In: *Proc. 17th Asia Pacific Software Engineering Conference (APSEC 2010)*. Sydney, Australia: IEEE, 2010, pp. 336–345. DOI: 10.1109/APSEC.2010.46.

[239] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. "Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus." In: *Software Engineering Notes* 38.5 (2013), pp. 1–4.

[240] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. "Recommending Refactorings to Reverse Software Architecture Erosion." In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 335–340. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.40. URL: http://dx.doi.org/10.1109/CSMR.2012.40.

[241] The R Foundation. *The R Project for Statistical Computing*. https://www.r-project.org/, Accessed October 2021.

[242] S. de Toledo, A. Martini, A. Przybyszewska, and D. Sjøberg. "Architectural Technical Debt in Microservices: A Case Study in a Large Company." In: *International Conference on Technical Debt (TechDebt)*. 2019, pp. 78–87.

[243] Michele Tomaiuolo, Federico Bergenti, Agostino Poggi, and Paola Turci. "OWLBeans - From ontologies to Java classes." In: *WOA 2004: Dagli Oggetti agli Agenti. 5th AI*IA/TABOO Joint Workshop "From Objects to Agents": Complex Systems and Rational Agents, 30 November - 1 December 2004, Torino, Italy*. Ed. by Matteo Baldoni, Flavio De Paoli, Alberto Martelli, and Andrea Omicini. Pitagora Editrice Bologna, 2004, pp. 116–125.

[244] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. "Design Pattern Detection Using Similarity Scoring." In: *IEEE Trans. Software Eng.* 32.11 (2006), pp. 896–909. DOI: 10.1109/TSE.2006.112.

[245] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. "How developers engage with static analysis tools in different contexts." In: *Empirical Software Engineering* (2019).

[246] Roberto Verdecchia, Philippe Kruchten, Patricia Lago, and Ivano Malavolta. "Building and evaluating a theory of architectural technical debt in software-intensive systems." In: *Journal of Systems and Software* 176 (2021), p. 110925. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110925. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000224.

[247] Roberto Verdecchia, Patricia Lago, Ivano Malavolta, and Ipek Ozkaya. "ATDx: Building an Architectural Technical Debt Index." In: *Evaluation of Novel Approaches to Software Engineering (ENASE)*. 2020.

[248] Tom Verhoeff. *From Callbacks to Design Patterns*. 2012.

[249] Santiago Vidal, Willian Oizumi, Alessandro Garcia, Andrés Díaz Pace, and Claudia Marcos. "Ranking architecturally critical agglomerations of code smells." In: *Science of Computer Programming* 182 (2019), pp. 64–85. ISSN: 0167-6423.

[250] Marek Vokac. "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code." In: *IEEE Trans. Softw. Eng.* 30.12 (Dec. 2004), 904–917. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.99.

[251] Edsger W. Dijkstra. "On the Role Of Scientific Thought." In: (Jan. 1974).

[252] Yaza Wainakh, Moiz Rauf, and Michael Pradel. "IdBench: Evaluating Semantic Representations of Identifier Names in Source Code." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 562–573. DOI: 10.1109/ICSE43902.2021.00059.

[253] Bartosz Walter and Tarek Alkhaeir. "The relationship between design patterns and code smells: An exploratory study." In: *Information and Software Technology* 74 (2016), pp. 127 –142. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.02.003.

[254] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. "Code smells and their collocations: A large-scale experiment on open-source systems." In: *Journal of Systems and Software* 144 (2018), pp. 1–21.

[255] Rongcun Wang, Rubing Huang, and Binbin Qu. "Network-based analysis of software change propagation." In: *The Scientific World Journal* 2014 (2014).

[256] Tao Wang, Gang Yin, Xiang Li, and Huaimin Wang. "Labeled Topic Detection of Open Source Software from Mining Mass Textual Project Profiles." In: *Proceedings of the First International Workshop on Software Mining*. SoftwareMining '12. Beijing, China: ACM, 2012, pp. 17–24. ISBN: 978-1-4503-1560-9.

[257] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. "Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree." In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 261–271. DOI: 10.1109/SANER48275.2020.9054857.

[258] P. Wendorff. "Assessment of design patterns during software reengineering: lessons learned from a large commercial project." In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. 2001, pp. 77–84. DOI: 10.1109/CSMR.2001.914971.

[259] U. Wilensky. *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999. URL: http://ccl.northwestern.edu/netlogo/.

[260] Martin B Wilk and Ram Gnanadesikan. "Probability plotting methods for the analysis of data." In: *Biometrika* 55.1 (1968), pp. 1–17.

[261] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. "Inter-smell relations in industrial and open source systems: A replication and comparative analysis." In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 121–130.

[262] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. "Automating change-level self-admitted technical debt determination." In: *IEEE Transactions on Software Engineering* 45.12 (2018), pp. 1211–1229.

[263] *YeD*. https://www.yworks.com/products/yed, Accessed October 2021.

[264] Robert K. Yin. *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*. 4th. SAGE Publications, Inc, 2009. ISBN: 9781412960991.

[265] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. "Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (2020), pp. 1145–1152. DOI: 10.1609/aaai.v34i01.5466. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5466.

[266] Franco Zambonelli and Andrea Omicini. "Challenges and Research Directions in Agent-Oriented Software Engineering." In: *Auton. Agents Multi Agent Syst.* 9.3 (2004), pp. 253–283. DOI: 10.1023/B:AGNT.0000038028.66672.1e. URL: https://doi.org/10.1023/B:AGNT.0000038028.66672.1e.

[267] Yu Zhang and Binglong Li. "Malicious Code Detection Based on Code Semantic Features." In: *IEEE Access* 8 (2020), pp. 176728–176737. DOI: 10.1109/ACCESS.2020.3026052.

[268] C. Zhu, X. Zhang, Y. Feng, and L. Chen. "An Empirical Study of the Impact of Code Smell on File Changes." In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2018, pp. 238–248.

[269] Ioannis Zozas, Apostolos Ampatzoglou, Stamatia Bibi, Alexander Chatzigeorgiou, Paris Avgeriou, and Ioannis Stamelos. "REI: An integrated measure for software reusability." In: *J. Softw. Evol. Process.* 31.8 (2019).

[270] hello2morrow. *Sonargraph*. www.hello2morrow.com, Accessed October 2021.

[271] I. Şora. "A PageRank based recommender system for identifying key classes in software systems." In: *10th Jubilee International Symposium on Applied Computational Intelligence and Informatics*. 2015.