DETERMINING THE RATIONALE OF ARCHITECTURAL SMELLS FROM ISSUE TRACKERS

THORSTEN RANGNAU - S3570282

university of groningen

faculty of science and engineering

Prof. Paris Avgeriou

Dr. Mohamed Soliman

27th August 2020

Thorsten Rangnau - s3570282: *Determining the rationale of architectural smells from issue trackers,* 27th August 2020

ABSTRACT

Architectural Smells are design fragments which can span over multiple components of a system's architecture and have system level impact. Thereby, they increase maintenance efforts and costs. This phenomenon is addressed by researchers who have focused on different smell types, smell detection mechanisms, or resolving techniques. So far, little is known about why architectural smells are incurred into software. However, understanding this can help to avoid adding smells into software. One way to find the rationales behind incurring architectural smells is to find versions in which new smells are added into the system and analyse the corresponding documentation available through issue repositories. Since architectural smells evolve over time, one difficulty in detecting a new smell instance is to detect all smell variations related to the same smell.

In this case study, six open source software projects are analysed. Together they comprise over 28,000 different software versions with over 62,980 smell variations related to 1,153 architectural smell instances. Analysing these instances allows to extract the motivation (e.g. Bugfix), the priority, and developer characteristics of the versions that incurred the smell. Furthermore, analysing discussions in documentation artifacts allows to understand the rationale behind incurring new smells.

The findings of this study show that smells can evolve in a tree-like structure. Additionally, the motivations behind incurring new smells seems to differ from project to project. Although, most smells are added through resolving an issue with priority level ‚Major'. Two findings suggest that developers are not aware of incurring new smells: (1) developers with the most changes incur the most smells, and (2) no discussions about smells during review processes. This implies that smells are incurred with a „hidden" trade-off between a quality (e.g. performance) and maintenance.

In conclusion, the findings suggest that developers are not aware of incurring architectural smells. In addition, the tree-like structure of the smell evolution provides new impulses for future research in this area.

Design today has reached the stage,
where sheer inventiveness can no longer sustain it.

— Christopher Alexander, 1963

ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. Paris Avgeriou and Dr. Mohamed Soliman for their great support during my master project.

I would further like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ADD Architectural Design Decision

AS Architectural Smell

LOC Lines of Code

TD Technical Debt

Part I

MASTER THESIS

1 INTRODUCTION

The idea that changes in software systems - applied over a longer time period - lead to difficulties in maintaining an application, is not a new one. In 1994, David Parnas already described the importance of keeping the development and maintenance costs resulting from *Software Aging* at a minimum level. Costs of maintenance increase with the size of the project because software often grows in complexity and degrades in system quality [30]. Potentially, this effect increases the number of new bugs introduced during incremental changes [42]. A more specific variation of the degradation phenomena is entailed in the notion of *Architectural Smells* (ASs). An AS is a design decision that negatively affects internal system qualities on an architectural level such as maintainability, extensibility, or adaptability [16]. Surveys conducted on their evolution revealed that in general ASs increases in number over the lifespan of the software system [36].

Means to address this issue are covered in the maintenance phase of a software project which involves refactoring actions with the goal to diminish the complications arising from increasing software complexity. Williams and Carver consider this phase of the software life-cycle as the most expensive one. It is therefore advisable to reduce needless sources of supplementary work (due to e.g. bad design practices) to a minimum in order to keep software maintenance manageable [42]. One field that attracts the attention of the software engineering community is the notion of *Technical Debts* (TD) which play an important role in managing software projects. TD describe software constructs that are expedient in the short term, but create a technical context that increases complexity and costs in the long term [2]. Hence, a software project can benefit from these constructs but also suffer from them if not addressed properly. Especially in the last decade, researchers and software practitioners alike increasingly show interest in this phenomenon and study the effects that incurring TD has on software maintenance. One crucial sub-topic for instance is technical debt management. Debts can occur on several levels of abstraction such as architecture, design, or implementation. ASs are a subcategory of TD [24] and this study focuses on them.

Garcia et al. found that ASs especially - but not only - affect inner system qualities [16]. Furthermore, those smells span over multiple components of the software architecture and have system level impact [35]. Sometimes engineers accept compromises by using TD deliber-

ately in order to achieve an urgent goal. However, undetected ASs in a software system are expensive. This is because resolving them (sometimes referred to as paying back the debt [11]) requires a lot of effort in identifying, understanding, and analysing them [9]. Therefore, principles and best practices are required to support software engineers in managing and resolving smells in their projects. This can help to keep the system under question in a maintainable state.

In order to support the management of technical debt, several studies are conducted so far with the focus on - among others - categorization of ASs, identification methods, resolving techniques, or evolution of smells. For instance, Garcia et al. argues that a catalog of ASs enables practitioners to detect existing smells in their architecture [16]. In [36], several characteristics about the evolution of certain smell types over time in a software system have been found. This supports prioritization of ASs in the technical debt management of a project.

However, all these techniques focus on removing existing smells form an architecture in order to establish better system qualities. Unfortunately, little is known about how to avoid the detrimental advent of ASs. We believe that with understanding why software engineers incur smells in a software system can help to slow down the degrading effect on software quality. We further understand that the causes for adding smells to a software system is captured in the rationale for the design decision that leads to the parturition of AS. Understanding the rationale will help software engineers to (1) avoid the inadvertently implementation of smells and (2) guide them in making trade-off decisions when deliberately introducing ASs towards a certain value in another quality dimension.

Once a smell is added into a software design, it creates a gap between a hypothetical ideal state of the system in which it can be maintained effortlessly and the current architecture [9]. In the context of ASs this means that the current architecture lacks the ability for maintaining and extending it to a certain degree. As already mentioned it is desirable to keep the number of smells to a minimum and thereby prevent the system's architecture to deviate too much from the hypothetical ideal state. Unfortunately, exact details on decisions for incurring ASs are unknown and the reasons for them are unrevealed. Nonetheless, these information can help to avoid this kind of TDs in a software system.

In the past two decades several studies have been conducted on decisions related to architectural design (e.g. [7, 21, 23, 38]). In fact, an AS is defined as an architectural design decision which impacts the internal system qualities directly [39]. In general, an architectural de-

sign decision is composed by a rationale and a consequence [38]. The rationale is the reason behind the design decision and describes why a change in the architecture is made [21].

In the mid 2000ers, Bosh and Jansen suggested Software Architecture as a set of architectural design decisions [21]. They described that architectural design decisions are composed by entities (including the rationale) that realizes one or more requirements on a given architecture. An important aspect is that a design decision can yield into new requirements that are in turn be addressed by new design decisions. In addition, each decision is also related to a change in the system's architecture. This creates a chain of decisions depending on each other where each also pinpoints to a change in the architecture.

This leads to the conclusion that an AS is a part of the decision chain since it is an architectural decision itself. Furthermore, it is composed by the same elements as a general design decision, including the rationale. This can help in identifying the reasons for incurring smells into a software system. Our perspective on the rationales for ASs is twofold. Imagine a requirement for a change of the architecture such as a new feature or a bug-fix. It can happen that the engineers encounter further requirements while making decisions on how to satisfy the initial requirement [10]. They address the new requirements with a new decision [21]. Imagine further, that this decision leads to changes in the architecture that add one or more ASs. Thus, several decisions exist that have an influence on why ASs are incurred in software. We divide these decisions into several kind of decisions. First, there are decisions that motivates the changes which eventually incur smells. Those motivations can be implementing a new feature or fixing a bug. These decisions influence ASs indirectly. Second, there are the actual decisions for adding the architectural constructs that form the smell. The second kind of decisions address ASs directly.

The aforementioned process repeats over the lifetime of the system for every architectural change. Therefore, a glance into the history of the system may help to reveal the rationales for incurring ASs. Essentially, this is what this study aims to analyse in the context of ASs. Therefore, we need to find a way to extract this information from available software documentation that directly points to the decision being made to introduce one or more ASs.

As Shahbazian et al. describe in [38], modern software development offers access to the history of a system via issue trackers and code repositories. In fact, studies have been already conducted where issue trackers such as Jira[1] were used in order to extract design decision [38,

1 https://www.atlassian.com/software/jira

39]. Apart from this, several tools are available that can detect ASs in a software project. A selection of such tools is listed in [3]. Some studies even focused in analysing the evolution of smells in a software project's history [13, 36]. However, neither of these works discussed the rationale for AS. As far as we know our study is the first attempt to combine the techniques mentioned above and thereby extract the reasons why developers incur ASs into their software systems.

As described in [38], the rationale for a decision is partially captured by issue items such as type or description in Jira. Therefore, it is our understanding that it is possible to extract rationales for decisions which motivates the changes that lead to new AS instances. We attempt to gain these motivating decisions from issue repositories following a quantitative analysis approach. Therefore, we first identify versions in which a particular smell instances is newly incurred. Subsequently, we can extract information about the motivation (i.e. issue type) or priorities for the changes of that versions from issue repositories. Finally, we can aggregate these information which will help us to unravel the reasons for incurring ASs via a quantitative analysis.

The rationale for adding the constructs that form the AS may be more difficult to extract. Several authors indicate that software engineers add ASs either intentionally or unintentionally to their projects. We were able to find evidence for the latter (e.g. [39]). However, we could not find any empirical evidence in previous work about deliberately incurring AS. Former studies have shown that software practitioners add code smells - another sub category of TD - deliberately in order to achieve value in a different dimension than internal software quality [6, 31, 34]. Along with this, several authors argue that the same is done with AS. Examples for this can be found in [9]. Yet, we were not able to find any proof for this in our literature research. In fact a former study on *Self-Admitted Technical Debt* (SATD) found four different types of debts in software systems not covering architectural smells [34]. Nonetheless, if ASs are introduced intentionally as a trade-off towards another system quality, then it would be interesting to study those scenarios in order to guide engineers in decision making processes as described above.

One way to preserve this knowledge is to use the issues related to a software version introducing ASs and extract the rationale for adding the architectural constructs forming the smell by analysing its comments. We find that developers sometimes use the comment function of issue trackers to discuss design and implementation decisions. With a qualitative analysis of these comments we expect to find (1) a first proof for deliberately implementing smells and if so (2) what

goals needed to be achieved with this decision.

With this study we contribute to enlarge the knowledge about the rationale for ASs. Therefore we analyse six open source *Java* projects from the *Apache Software Foundation* (ASF)[2]. The source code of all these projects is available on GitHub[3] and their corresponding issues are managed using Jira. The source code is analysed utilizing the existing smell detection tools Arcan [13] and ASTracker [36]. This facilitates the identification of software versions in which a new AS instance is added. Subsequently, it is possible to connect this version to the corresponding issue in Jira. In fact, this is where the actual empirical study of our work begins. We aim to quantitatively achieve information that can help us to find the reasons for adding ASs into software.

We further use these issues to qualitatively analyse the discussions of the developers entailed in each issue. This helps us to identify the rationale for adding a certain architectural fragment that incurs a new AS instance. In particular the qualitative analysis shows whether software practitioners introduce smells deliberately or not and if so what are the benefits of this trade-off decisions.

Finally, the results of our study show that there are certain situations in which ASs are incurred more likely. However, the motivations for this cannot entirely be explained by issue types or priorities. The quantitative analysis showed that there is no proof for AS instances that have been incurred deliberately. In addition, some smells are unintentionally incurred in order to improve a certain system quality such as performance or security.

The remainder of this thesis is structured as follows. Section 2 depicts important background information, whereas literature related to this study is discussed in Section 3. The design of the empirical case study and the research questions are presented in Section 4. The findings of this thesis are presented in Section 5 and discussed in Section 6. Subsequently, we present threats to validity in Section 7 and finally, Section 8 concludes the thesis and outlines suggestions for future work.

2 https://www.apache.org/
3 https://github.com/

2

BACKGROUND

This section discusses background information that make up the foundation for the research conducted in this study. Therefore, we present the fundamentals of TDs in Section 2.1. Subsequently, we describe the notion of dependencies and stability among packages in Section 2.2 which form well known principles in architectural design. Furthermore, we explain the architectural smells (AS) and their subcategories in Section 2.3. We especially, delve into instability architectural smells which are based on the notion of stability. We also discuss the impact of this specific smell category on maintainability because our case study has a focus on these smells. Finally, Section 2.4 provides an overview of the concepts of architectural design decisions and how this is related to our work.

2.1 FUNDAMENTALS OF TECHNICAL DEBT

The concept of *Technical Debts* (TD) was coined by Ward Cunningham in 1992 to describe the need for a refactoring of a system in order to address non-functional aspects to non-technical stakeholders [11, 29, 37]. This definition was later refined by the participants of the *Dagstuhl Seminar 16162* about the management of TD in *Software Engineering*. They understand that going into TD is when software developers make technical compromises which are expedient in the short term, but lead to increasing complexity and costs in the long term [2]. They further discovered that TDs are mostly incurred unintentionally and that it mostly has a negative impact on maintainability and evolvability.

Martin Fowler specified the *Technical Debt Quadrant* in order to demonstrate possible scenarios derived from his experience why developers go into TDs in software projects [15]. The quadrant is divided into two columns (*Reckless* and *Prudent*) and two rows (*Deliberate* and *Inadvertent*) resulting in four cells. The *Reckless/Deliberate* cell describes situations in which software teams are aware of breaching good design practice. Yet, they believe they cannot afford spending time on following overall accepted design principles. Other software practitioners may be ignorant of those principles acting reckless and are even unaware the they going into debts with their approach (*Reckless/Inadvertent* cell). There may be development teams that are aware of breaching design or implementation principles but nonetheless go into TDs to for example meet a certain release date. They know that

they eventually have to pay back the debt and may already spend some thoughts about how to resolve these problems. This scenario is addressed by the *Prudent/Deliberate* cell. Finally, the *Prudent/Inadvertent* cell describes excellent software designing teams. They tend to always create the best possible code but still find design flaws after some reflection time. Following Fowler[15], this is inevitable since developing software is to always keep up learning and improving the own skills. Figure 1 shows the technical debt quadrant by Fowler.

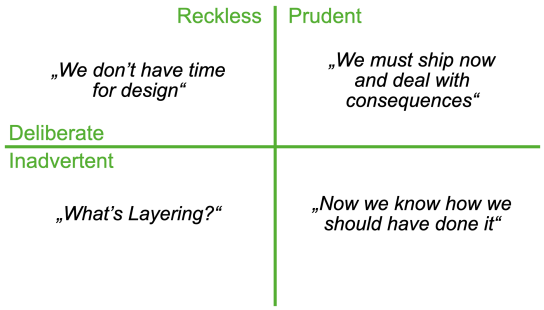


Figure 1: Technical Debt Quadrant by *Martin Fowler* [15]

Ozkaya, *Nord*, and *Kruchten* mentioned that several authors have been trying to divide technical debts into subcategories [29]. One problem that occurred with this is that the use of the TD metaphor was exaggerated. In order to clearly differentiate between technical debts and other issues of software maintenance management, they created the technical debt landscape shown in Figure 2. The landscape distinguishes between visible and invisible elements. Visible elements are those that add new functionality or fix bugs. The invisible elements concern the changes necessary to bring the system from the current state to a hypothetical ideal state in terms of maintenance. They named those elements as "invisible" to emphasis that the necessity for these changes can be only seen by software engineers but not by non-technical stakeholders. Furthermore, the invisible elements facilitate the TDs.

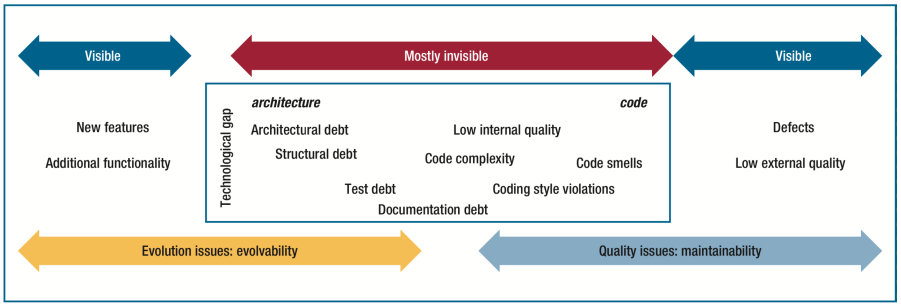


Figure 2: Technical Debt Landscape by *Ozkaya*, *Nord*, and *Kruchten* [29]

After delimiting the term TD from other software issues, one can subdivide debts into a collection of different TD types. These types encompass and describe different levels of abstractions where debts can occur. For example *Li*, *Avgeriou*, and *Liang* classified TD types into ten coarse-grained categories [24]. The types comprise of *Requirements TD*, *Architectural TD*, *Design TD*, *Code TD*, *Test TD*, *Build TD*, *Documentation TD*, *Infrastructure TD*, *Versioning TD*, and *Defect TD*. Each of those types can be further organized into sub-types. This study focuses on the specific sub-type of *Architectural TD* and hence only these sub-types are discussed in this study for the sake of brevity[1].

Architectural TDs have six sub-types, namely *architectural anti-patterns*, *complex architectural behavioral dependencies*, *violations of good architectural practices*, *architectural compliance issues*, *system-level structure quality issues*, and AS [24]. In correspondence with [27], this leaves us with the conclusion that ASs are a subcategory of TDs. Categorizing ASs as TD enables us to apply the same characteristics to both.

Finally, we want to provide a demonstrating example that explains incurring debts on architectural level. At the beginning of this thesis in Section 1, we described that software aging degrades a system's quality which is also known as architectural drift or system erosion [41]. In this case, the architecture often deviates from its original intent and makes maintenance more time consuming and more expensive, while the reliability of a system decreases. This is due to design decisions made without considering the impact on inner system qualities [27]. Then, a software system incurs a TD on architectural level and such a short-term compromise can lead to significant long-term problems. If not resolved, this effect gets stronger and the system gains more quality or maintainability interest on its debt.

2.2 FUNDAMENTALS OF STABILITY

In 1994, Martin analysed the quality of object oriented design [25]. In particular he focused on the dependencies among subsystems of the design in order to find what makes a systems more robust, reusable, or maintainable. He identified stability of the dependencies as crucial factor in achieving a certain quality of a system in these dimensions. Furthermore, Martin defined stability as the likelihood of a change in a subsystem where another subsystem depends upon. In addition, he described a metric called *Stability Metric* in order to determine the stability between subsystems.

1 for a complete list of all TD types and sub-types please refer to [24]

Moreover, Martin introduced stability as a mean to determine how packages of a software system should be interrelated [26]. It uses the relationships among packages, also known as dependencies, to define principles that help to improve the structure of an application in order to keep it in a maintainable state. Designing a system by following these principles, reduces problems that usually occur when software teams work in parallel on different parts of the system.

Consider that “bad relationships" among packages result in a rigid structure of an application. A rigid structure is a structure that is hard to change and hence undesired [25]. This is due to fact that a single change in such a heavily interdependent software design may influence other parts of the system as well. As a consequence those parts have then also to be changed. In other words, such a system has the tendency to break in many places when a single change is made.

2.2.1 *Dependencies*

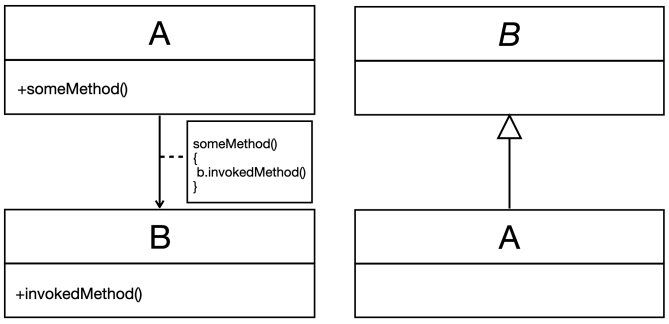
In order to understand these “bad relationships" among packages one needs to first understand their origins. In an object-oriented programming language such as Java[2], relationships can be formed only among classes. They are called dependencies (D), defined as follows [14], and illustrated in Figure 3:

D1 **Dependencies between classes:** Class A invokes a method of class B where the direction of the dependency goes from class A to class B (A depends on B) as shown in Figure 3a.

D2 **Hierarchy dependency:** The child class A depends on the parent class B where the direction of the dependency goes from class A to class B (A depends on B) as shown in Figure 3b.

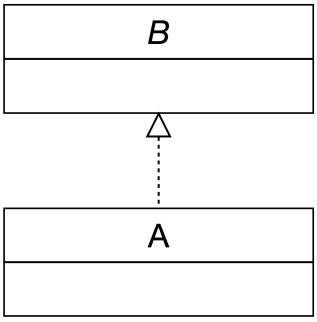
D3 **Interface dependency:** The implementing class A depends on the interface B where the direction of the dependency goes from class A to interface B (A depends on B) as shown in Figure 3c.

2 In this study we focus only on Java projects - see more in Section 4



(a) Dependencies between classes.

(b) Hierarchy dependency.



(c) Interface dependency.

Figure 3: Dependencies between classes.

Those dependencies can occur between classes of the same package or between classes of different packages. The latter cases are important to understand the principles of package coupling. There are two dependencies among packages important for this study [14, 25]:

AD: **Afferent Dependency:** *Package a* contains class A and *package b* contains class B. Class B depends on class A. Thus, *package a* has an afferent or incoming dependency coming from *package b*. The direction of the package dependency is from *package b* to *package a*. Figure 4 demonstrates an afferent dependency of *package a*.

ED: **Efferent Dependency** *Package a* contains class A and *package b* contains class B. Class A depends on class B. Thus, *package a* has an efferent or outgoing dependency going to *package b*. The direction of the package dependency is from *package a* to *package b*. Figure 5 shows the efferent dependency of *package a*.

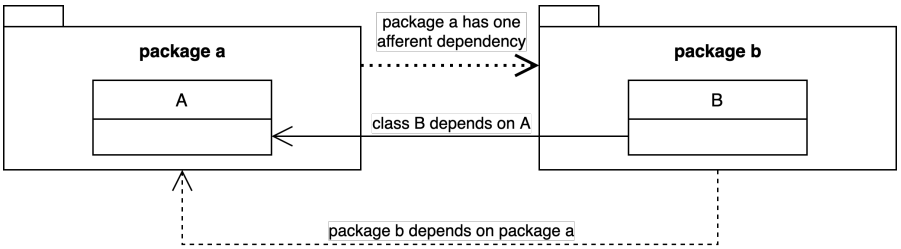


Figure 4: Afferent dependency of package a.

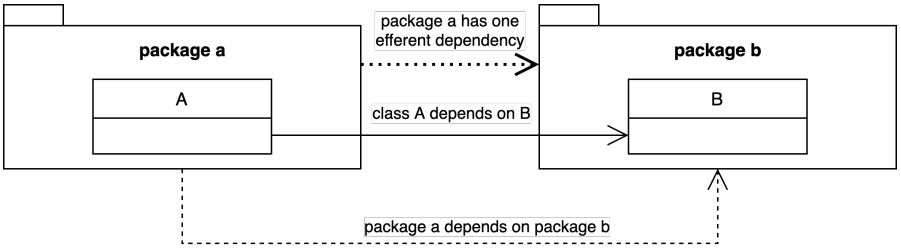


Figure 5: Efferent dependency of package a.

The dependencies among all packages create a graph which is called the *package-dependency graph* where the packages are the nodes and the dependencies are the edges [26].

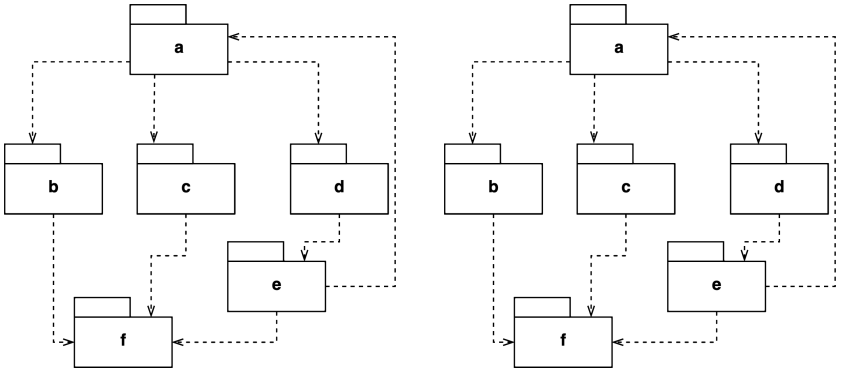
2.2.2 *Principles of Package Coupling and Dependency Management*

In order to keep a software system in a maintainable state, Robert C. Martin introduced two principles related to dependencies. The smells that we analyse in this study are basically violations of these principles. In order, to understand them we provide a brief description of the two principles.

> **Acyclic-Dependency Principle (ADP)** - The Acyclic-Dependency Principle says:
>
> *Allow no cycles in the package-dependency graph* [26].
>
> One problem that occurs in software projects where multiple changes are made in parallel is the *Morning-after Syndrome* [26]. It describes the situation where developers are in believe that all problems related to their work are fixed when they finish in the evening. However, the next morning everything is broken. This is because other developers made changes to another part of the system over night where the former part was dependent on.

In order to avoid these situations it is crucial to keep the number of dependencies a package that needs to be changed has as minimal as possible. However, when there is a cycle in the dependency graph, packages in need of a change might depend on more packages than necessary. This leads to a higher maintenance effort.

Consider a packet structure as displayed in Figure 6a. *Package e* only depends on *package f*. Hence, if someone makes changes in *package e*, she only needs to check whether there are also changes in *package f* and tests accordingly. In contrast to this, consider the packet structure in Figure 6b. Here *package e* also depends on *package a* which forms a cycle since *package a* depends on *package d* which depends on *package e*. In addition, *package a* is direct and indirect dependent on all other packages of the application. One making changes to *package e* has also to check check and test against all other packages. The effort for maintenance has increased with the cycle. This effect exacerbates the more complex the system becomes and hence cycles should be avoided.



(a) Acyclic packet structure. (b) Cyclic packet structure.

Figure 6: Differences of cyclic- and acyclic package structure.

Stable-Dependency Principle (SDP) - The Stable-Dependency Principle says:

Depend in the direction of stability [26].

Martin describes stability as a characteristic of a an item that is related to the amount of effort that is required to change it [26]. The more stable something is the more work is needed to make a change on it. As implied in the description of the previous principle, it is easier to change a package with no or only few dependencies to other packages than to a package with a lot of

afferent dependencies. Following this, a package with multiple incoming dependencies is a stable package since it takes a lot of effort to change it (Figure 7a). Contrary to this, an unstable package is a package that only has efferent dependencies (Figure 7b).

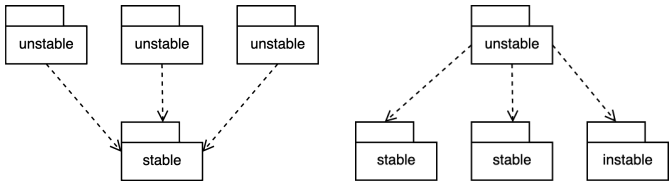
Some packages are more stable than others. The *Stability Metrics* help to compare the degree of stability of all packages in the application as defined by [25]. Definition 1 to 3 define the metrics. Most important is the *Instability Metric* of Definition 3. It can be calculated using Equation 1. The metric has the range from [0, 1] where $I = 0$ indicates a maximally stable package and $I = 1$ a maximally unstable package.

Definition 1 C_a*: Afferent Couplings: The number of classes outside this package that depend on classes within this package.*

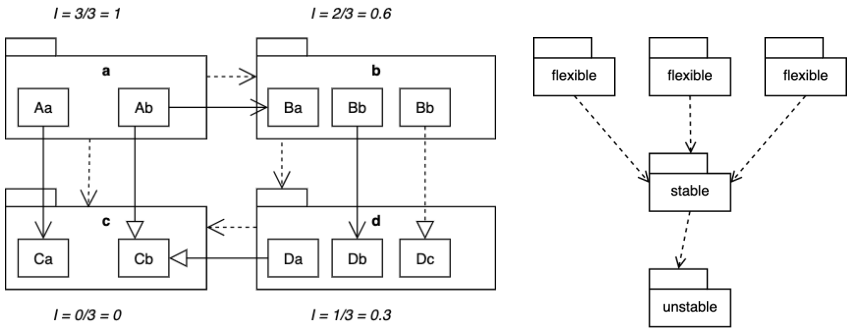
Definition 2 C_e *Efferent Couplings: The number of classes inside this package that depend on classes outside this package.*

Definition 3 I*: Instability: Index that shows the positional stability of this package. It is is calculated by Equation 1:*

$$I = \frac{C_e}{C_a + C_e} \tag{1}$$



(a) Stable dependency. (b) Unstable dependency.



(c) Example of instability metrics. (d) Violation of SDP.

Figure 7: Stable and unstable dependencies following [26]

The package structure depicted in Figure 7c may serve as an example for clarification. There are four packages containing a various number of classes. The classes have dependencies to other classes outside of their packages (arrows with solid line). Using Equation 1 one can calculate the *Instability Metric* for every package. *Package a* has three outgoing dependencies and no incoming one. Applying equation 1, it has an *Instability Metric* of $I_{\text{package a}} = \frac{3}{0+3} = 1$. This package is very unstable and can therefore be easily changed without any side effect on other packages. Calculating the *Instability Metric* for the other packages in the same way derives in the following results: $I_{\text{package b}} = 0.6$, $I_{\text{package c}} = 0$, and $I_{\text{package d}} = 0.3$. We can now say that *package c* is the most stable package and that *package d* is more stable than *package b*. We already know that *package a* is the package with the least stability. In addition, a closer look at the package dependencies (arrows with stressed line) reveals that this example follows the SDP since each package depends on a package that is more stable than itself.

The previous example showed that not all packages are stable. Indeed, this is not desirable especially when the system should be changed effortlessly for e.g. maintenance purpose [26]. Therefore, the design of the system needs to address packages that are required to be changed easily and make them as unstable as possible.

In Figure 7d we provide an example of a package structure that violates the SDP. The package in the middle has three other packages that are depended on it. It is the most stable package in this example and thereby requires a lot of effort in changing it. Consider that the package at the bottom is one that needs to be flexible (unstable). Its incoming dependency from the stable package however makes it hard to change. This is a clear violation of the SDP and the designers of this system disregarded that their decision makes it harder to maintain the system.

2.3 FUNDAMENTALS OF ARCHITECTURAL SMELLS

The one type of Architectural Technical Debt this study is focusing on is the poignant combination of software architecture constructs called Architectural Bad Smells or simply Architectural Smells. An AS is a design decision that has a detrimental impacts on the quality of a software system [16]. Software engineers can incur an AS intentional or unintentional. However, the resulting architecture of that decision is not faulty but rather expedient towards the functionality [17]. The

negative effect of the smell focuses on the internal software qualities such as maintainability or extendability.

As the name suggests, ASs span over multiple components of a software project and have system level impact [35]. The components involved engender architectural constructs whose characteristics reduce the ability of maintaining the system such as adding new features or sustainably fixing defects or alike [17]. One may imagine an "hypothetical" ideal state of the system in which common design rules were applied. Incurring ASs leads to a deviation of the system's architecture and this ideal state [9]. This inevitably creates a gap between desired and existing architecture. Furthermore, as the complexity of a system increases over time, so will the number of ASs [36].

The impact of the smell is always on inner qualities. Nonetheless, this does not necessarily apply only to maintainability or extensibility but can effect other qualities such as performance or security as well [17]. As a consequence, undetected ASs are expensive because paying back the debts requires huge effort in identifying, understanding, analysing activities [9]. These are required in order to thoroughly and sustainably remove the smell from the project. This is eventually necessary in order to keep the system in a maintainable state.

There are two main reasons for the origins of an AS [9]. The first one is due to the structural complexity a system achieves during its lifespan. This is often inadvertently introduced as the number of dependencies between system components grows and design goals are violated. The second one may be that the team expects some kind of strategic value from deliberately violating design principles. This can also be understood as a trade-off decision towards another quality dimension but against inner system qualities [16]. A thinkable situation may be that the development team is eager to meet a certain release date which may be easier to achieve without following the rules of good software design. These two origins of AS follow the column dimension of the technical debt quadrant by Martin Fowler (see Section 2.1).

2.3.1 *Instability Architectural Smells*

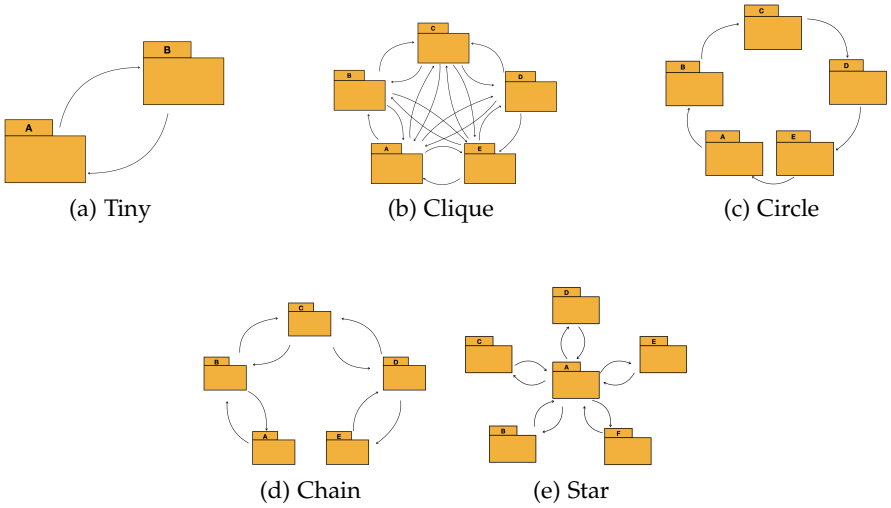
Three specific smells that got the attention of the research community are called *Instability Architectural Smells*[3]. This group of ASs are named after the instability metrics defined by *Martin* [14, 25] which are described in Section 2.2. These metrics describes whether a package of a software application can be changed without impacting other packages within the same application [14]. All three ASs that we de-

3 We will address those also as ASs

scribe in this section are related to these metrics. The smells are cyclic dependency, unstable dependency, and hub-like dependency. We describe them in detail in the following sections.

2.3.2 *Cyclic Dependency*

The *cyclic dependency* is an AS that violates the ADP [40] which is described in Section 2.2.2. This smell composes of a subsystem that forms a chain of dependencies that break the desired acyclic nature of the dependency structure [14]. *Al-Mutawa et al.* conducted a research on the shape of cyclic dependency. They identified two groups of shapes (symmetric and asymmetric). Figure 8 and 9 provide an overview of the different shapes [1].



(a) Tiny (b) Clique (c) Circle

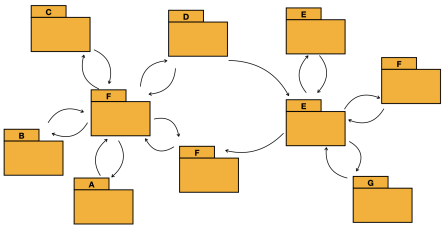
(d) Chain (e) Star

Figure 8: Symmetric shapes of cyclic dependencies following [1]

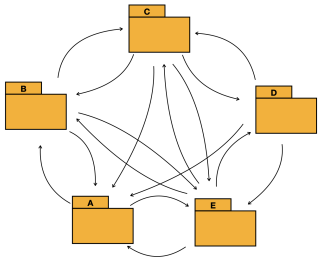
2.3.3 *Unstable Dependency*

The *unstable dependency* occurs when a component or package of a system depends on a another component or package that is less stable than itself [13]. This is a clear violation of the SDP which is discussed in Section 2.2.2. Figure 7d illustrates an unstable dependenc. As already mentioned, the stability of a component derives from the number of dependencies the component depends on and the number of other components depending on it. Imagine three components depend on *component A*. Then, a change in *component A* may have affects on the three other components. Therefore, *component A* is stable and hence requires a higher effort in changing it. If now *component A* depends on *component B* which is an - maybe by intention - unstable component, each change in *component B* may influence *component A* may influence the three former packages. This structure has a detri-

mental impact on the inner quality attributes of the system and is hence an AS.



(a) Multi-hub



(b) Semi-clique

Figure 9: Asymmetric shapes of cyclic dependencies following [1]

2.3.4 *Hub-Like Dependencies*

This smell arises when a component has a large number of efferent and afferent dependencies [40]. Figure 10 depicts this smell. As one can see, there are several spots that are hard to change in this smell. First, *package a* may be very hard to change since it is a stable package where each change may influence on of the depending packages. Similar to this, changing on of the packages *package a* relies on, may result in the same problem.

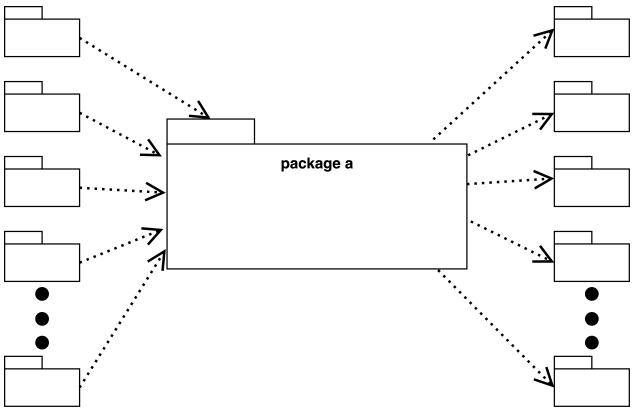


Figure 10: Hub-Like dependency on packages where *package a* is the hub

2.3.5 *Tool Support*

There are a lot of tools available that support the detection of ASs [3]. In this section we present the tools that we use in this study.

Arcan - Arcan is a tool developed by academia in order to detect and report architectural anomalies [13, 14]. It can parse *Java* source code and detect the three ASs that we described above. Theses types are, cyclic dependency, unstable dependency, and hub-like dependency. Arcan makes use of graph technologies that helps with representing the dependency graph of an application. The detection process includes reading of the Java bytecode, analysis and graph generation, architectural smell detection, and output. In the bytecode Rrading phase the precompiled files from the given Java application are read. This allows to extract content information about the structure of the project. Subsequently, this information is used for dependencies analysis and dependency graph generation. The generated graph is then used in order to identify the specific ASs. Once detected the information of the smells are stored in a graph file.

Apart from the process description found in literature, our work with Arcan revealed that this tool is also capable in parsing uncompiled *Java* projects. In addition, we also found that Arcan is able to use a Git repositories and create a graph file for every single software version of a project.

ASTracker - ASTracker[4] is a publicly available application developed by *Darius Sas* for the research on AS evolution [36]. It uses the graph files generated by Arcan for each version and maps every smell in each version to its closest successor in the next version. It also calculates the new smell characteristics and alike. Finally, ASTracker generates CSV or graph output files.

Designite - Designite is a commercial tool to identify TD in a software project[5]. It can detect a lot of different types of ASs [3]. This includes , unstable dependency, and hub-like dependency. Compared to Arcan, Designite uses slightly different detection algorithm to detect smells in the source code. Nevertheless, we use Designite to evaluate detected smells. Unfortunately, as far as our knowledge is, Designite is not able to parse multiple versions of a project as Arcan is capable of. For this study we use a free academia version available for students and teachers.

4 https://github.com/darius-sas/astracker
5 http://www.designite-tools.com/

2.4 FUNDAMENTALS OF DESIGN DECISIONS

As mentioned earlier, an AS is an architectural design decision. Therefore, we provide crucial information about these decisions in this section.

2.4.1 *Architectural Design Decisions*

Jan Bosch defined Architectural Design Decisions (ADD) as follows [7]:

> We define an architecture design decision as consisting of a restructuring effect on the components and connectors that make up the software architecture (and resulting system) as a consequence of the design decision, design constraint imposed on the architecture and a rationale explaining the reasoning behind the decision. In our definition, the restructuring effect includes the splitting, merging and reorganisation of components, but also additional interfaces and required functionality that is demanded from components.

In addition, an ADD is the outcome of a design process during the evolution of the system [21]. This corresponds to the definition of AS in Section 2.3 as these smells are defined as ADDs with a negative impact on system qualities. In other words an AS is a detrimental ADD.

The structure of an ADD composes of a rationale, design rules, design constraints, and additional requirements [21]. In addition, Shahbazian et al. added a consequence which is a description of the changes in the architecture the decision is resulting in [38]. These features can be mapped to ASs where the rationale is the reason for the smell. Design rules in an AS decision reflect in the violation of design principles such as the ADP or SDP. The design constraints manifests in the reduction of inner system qualities that constrain e.g. maintenance of the system. The additional requirements represents the required means of TD management derived by introducing this smell. Finally, the consequence of the smell is that subsystem that forms the structure of the implemented AS.

ADDs can be divided into four different kinds of decisions called Existence Decisions or Ontocrises, Ban/Non-existence Decisions or Anticrises, Property Decisions or Diacrises, and Executive Decisions or Pericrises [23]. The Ontocrises adds structural or behavioral elements to the system. Its consequence results in the most visible artifacts of the architecture such as components and connectors. The Anticrises restricts elements from being present in the design or implementation.

Those elements are very hard to detect since they have no manifestation at all in the system. Diacrises are decisions that are equally hard to be traced. This decision type refers to overarching traits or system qualities. The last ADD type, the Pericrises encompasses decisions that are only implicitly connected to design elements or their qualities. It captures decisions concerning financial, methodological, or educational means with an impact on developing the system. Aligning ASs to one of these types is not as straight forward as it may appear at the beginning. Technically, a smell adds a construct of design elements to the architecture (Ontocrises). However, by changing the structure of the system, ASs also affects the inner system quality (Diacrises).

Philippe Kruchten describes that ADDs have a certain state during the design process [22]. For example, a design decision can start with an idea for how to satisfy a requirement, may become tentative during discussions, and finally be decided and approved by the development team. It is hard to include the smell decision in this schema because we believe that the decision states, described by*Kruchten*, refers to intentional decisions. Yet, we were not able to find proof that ASs have been deliberately incurred into software architecture.

Contemplating the evolution of software architectures, reveals that their structure changes over time. These restructuring effects are the implications of ADDs since each decision changes the structural elements [7]. These structural changes affect also system qualities directly [38, 39]. This is a general explanation of ASs as they (1) adding structural complexity to the system and (2) change the inner system qualities negatively.

Another key issue and part of several studies is the way of documenting ADDs. In [23], Kruchten, Lago, and van Vliet describe four documentations practices for ADDs and their rationales:

1. **Implicit and undocumented** - being unaware of decisions or dismisses them as “of course knowledge"
2. **Explicit but undocumented** - specific reason for decision but remains undocumented
3. **Explicit but explicitly undocumented** - specific reason for decision but (tactically) hide reason
4. **Explicit and documented** - specific reason for decision and documented to preserve knowledge

2.4.2 *Rationale*

This study focuses on the rationale for ASs. Since these smells are ADDs as explained in Section 2.4 and each decision has a rationale, we provide useful information on them in this section.

The rationale is the justification for an ADD which adds some value to why the decision was made [21, 23]. There are two facets of rationales: intrinsic and extrinsic. An intrinsic rationale is the property of an ADD and hence belongs to the very same. An extrinsic rationale represents the relation of one decision to another.

There are multiple relations between ADDs resulting in a graph-like structure. Jansen and Bosch for example described that a decision for using a certain technology results in a decision of how to implement that technology in the system under question [21]. A list of different types of relations can be found in [23]. For this study these different types are not important but the fact that there is a relation between decisions is. Considering multiple decisions related to a decision for a smell entails in a decision chain that relates multiple extrinsic rationales to this smell decision.

Following this, we present one example that clarifies the affiliation of smells to its extrinsic rationale. Assume that fixing a bug in a software system results in an AS. This smell is an ADD and hence has an intrinsic rationale (for instance: inadvertent, deliberate, and so on, see Section 2.1). However, it also has one or more incoming extrinsic rationales that relate the decision incurring the smell with the decision to fix the bug. Hence, fixing the bug becomes the extrinsic rationale for the smell. Similarly, the decision which developer is working on resolving the issue is also connected to the decision that incurs the smell.

Putting all information together Figure 11 provides an overview of such a decision chain which results in incurring an AS. As one can see there is a delta that represents the changes in the system. These changes are the result of the corresponding decision (bigger green circle). The delta entails several smells that eventually incur a smell. Furthermore, we can see that the decision of for the changes has a rationale (bigger yellow box). Besides this there are other, preceding decisions in the figure (D1 and D2) each with its corresponding rationale. One example could be that D1 represents the decision for a change where the rationale is fixing a bug. This leads to the D2 decision which states that a certain developer should fix this bug. The rationale for this may be that the selected developer has good knowledge of that part of the system where the bug is present. This devel-

oper than makes the changes and - intentionally or not - decides to incur an AS. This example may serve as a demonstration how decisions and their rationale can influence the creation of a smell instance.

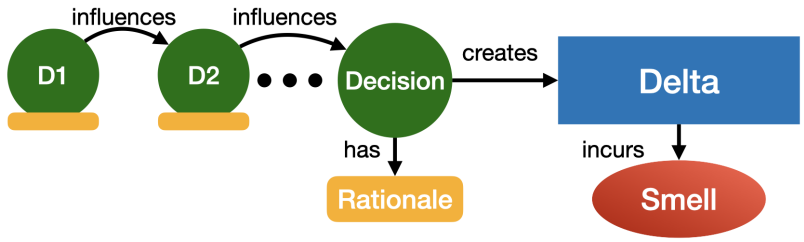


Figure 11: Overview on the structure of ADDs incurring an architectural smell

3

RELATED WORK

This section presents research related to our work. Since this study relies and combines methods and findings form different fields of software engineering we describe those areas separately. Therefore, Section 3.1 describes related work on *Architectural Smells*. In Section 3.2 we outline the previous work on *Architectural Design Decision*. Finally, Section 3.3 provides an overview on *Self-Admitted Technical Debt*.

3.1 RESEARCH ON ARCHITECTURAL SMELLS

Garcia et al. identified certain architectural structures in software with a negative effect on inner system qualities - maintainability, extensibility, adaptability, and so on - in 2009 [16]. They noticed frequently recurring design fragments recovered from architectures of existing systems which they called *Architectural Bad Smells*. The method used in this study was to analyse existing software systems through reverse-engineering the architectures of eighteen software systems. With their work they (1) introduced and defined the notion of ASs and (2) described four smell types which they had discovered during their research (*Connector Envy*, *Scattered Functionality*, *Ambiguous Interface*, and *Extraneous Connector*). In addition, they have not only provided a description of the identified smells but also defined quality impacts and trade-off descriptions for them. All in all, this research enables software architects to detect ASs in software projects and asses the corresponding impact on relevant qualities. They later refined their work in [17] and demanded further research on categorization, detection, and resolving methods of ASs. However, Garcia et al. do not describe an approach in how one can detect smells in existing software. Additionally, they did not investigate on the reasons why ASs are incurred into software.

In his work *"Agile Software Development: Principles, Patterns, and Practices"* [26], Martin consolidated his research on dependencies among subsystems and defined principles in software design resulting in a higher stability of the application. Those are the *Acyclic-Dependencies Principle*, *Stable-Dependencies Principle*, and *Stable-Abstraction Principle*. These principles can help software practitioners to design an application with a certain quality among inner system qualities. Although this work is not about ASs, it is the foundation for a specific group of smell types called *Instability Architectural Smells* which most of the literature presented in this section has the focus on. Therefore, this

work is different from our work because it does not address the rationale for ASs. In addition, it provides theoretical background on software design derived by the experience of the author but not an empirical case study.

In 2016, Fontant et al. picked up on the notion of ASs and combined it with the instability metrics defined by Martin [14]. Thereby, they introduced three specific ASs which they called *Instability Architectural Smells*. Those smells describe violations of the three principles by Martin, namely *cyclic dependency* , *unstable dependency*, and *hub-like dependency*. Besides their theoretical work, they presented the ASdetection tool Arcan which is capable of identifying their three presented ASs in Java source code. The tool has been tested on seven open source projects and the existence of the smells has been compared with results of other tools for the smell types cyclic dependencies and unstable dependencies.
One year later, this work was refined and detection algorithms and performance of Arcan was improved in order to provide better support for software teams to automatically detect smells in their projects [13]. The new Arcan version was tested against the two software projects *DICER* and *Tower4Clouds*. The results were manually verified with a precision of 100% for both projects and a recall of 60% and 66%, respectively.

Sas, Avgeriou, and Fontana conducted research on the evolution of instability ASs in 2019 [36]. They reused the Arcan tool in order to detect all smells for each version of a software project. In order to combine the results and track the smells of each version, they developed a tool called ASTracker[1]. With this they analysed 14 different open source *Java* projects. Their findings suggest that the three ASs types cyclic, unstable, and hub-like dependencies diverge in multiple aspects such as growth rate or time each smell instance affect the system. These insights led to the conclusion that maintenance effort should focus first on resolving unstable and hub-like dependencies. Cyclic dependencies - especially when recently introduced - are very likely to disappear within the next few releases.

In 2019, Fontana, Azadi, and Taibi compared nine AS detection tools provided by academia and industry [3]. In their work they presented a catalogue of smells where detection of each smell is guaranteed at least from one tool. For each smell they discussed the different approaches each tool follows in order to detect that particular smell. They encountered a lack of standardization manifesting for example in different naming for the same problem (e.g. hub-like dependency is also known as hub-like modularization or link overload). The com-

1 https://github.com/darius-sas/astracker

parison of the tools revealed that the tool Designite [2] can detect the most smell types. Finally, they classify architectural smells in order to enable intuitive understanding of an AS and in order to get a better idea of refactoring methods.

This study is based on the findings and insights of the aforementioned work. We use the detection mechanisms and tools presented in [13] and [36]. We further use the tool Designite in order to validate smells detected in a particular version. Nonetheless, our study is different from the studies presented in this section since their focus is on describing, detecting, and thus supporting refactoring methods of ASs. The aim of our study is to apply an empirical analysis in order to find the rationales for introducing ASs into a software system. As already mentioned, we believe that this will help to avoid implementing smells on the one hand but also guide trade-off decisions when deliberately add smell.

3.2 RESEARCH ON ARCHITECTURAL DESIGN DECISIONS

As mentioned in Section 2.3, ASs are defined as an *Architectural Design Decision* with a negative affect on inner system quality. We understand that a smell is formed by the restructuring effects on the architecture of that decision. It is hence the consequence or solution of the decision for incurring (deliberately or inadvertent) ASs. With this study we aims to extract the rationales for this decision. Rationales are a crucial part of design decisions and it is hence helpful to discuss previous work on design decisions.

In 2004, several software architects became aware that the traditional view on software architecture depending on the key concepts of components and connectors suffers from a number of key problems that lead to high maintenance cost and violations of design principles. Therefore, some architects suggested to extend this traditional view and consider *Architectural Design Decisions* (ADD) as part of software architecture [7, 21, 22]. In 2004, Jan Bosch described the problems of the traditional view of software architecture at this time in a position paper [7]. He claimed that the software design community need to take the next steps and adopt the perspective that software architecture is a composition of ADD. He further identified four relevant aspects of such a decision: *restructuring effect*, *design rule*, *design constraint*, and *rationale*. In contrast to our work, Bosch describes a general view on design decisions. In our work we have the focus on the rationale for ASs. Furthermore, Bosch suggests a paradigm change for describing software architecture. In our study we conduct an em-

2 http://www.designite-tools.com/

pirical case study in order to derive the rationales for ASs.

In the same year, Philippe Kruchten picked up the idea of representing ADDs as first-class entities of software architecture and suggested a design decision model [22]. This model distinguishes several kinds of decisions, describes their attributes, and defines relationships between them. Kruchten stated that this model should enable architects to better include their decisions into the architecture of complex software-intense systems and allow to create decision graphs provide an overview of interrelated design decisions. This will then support support reasoning about this high level abstraction of software architecture. Similarly to Bosch, Kruchten provides a model that can be used to extract information on design decisions. However, as well as Bosch, this work describes a model to capture software architecture related information. It does not conduct an empirical study on the rationale of ASs.

In 2005, Bosch and Jansen presented a new perspective on software architecture describing the architecture of a system as a composition of a set of ADD [21]. This perspective makes decisions an explicit part of software architecture. They claim that the resulting *Archium* model reduces knowledge vaporization and thereby alleviating software erosion. The model composes of the architectural, design decision, and composition sub-model. Thus, it is possible to describe an architecture using the notion of deltas (representation of a change), design fragments, and design decisions (entailing also but not only the rational). The notion of deltas implies that a decision can lead to another decision. In fact, the example provided in their work explicitly describes such a situation where a decision for integrating an existing anti-fraud system as a subsystem into the system under question is followed by a decision of how this integration should be done. The practicality of the model is demonstrated and validated by applying it to a case scenario. While an AS is incurred by a delta and information on the rationale can thereby be extracted with the *Archium* model, this study does not focus on finding rationales for AS. Although, Bosch and Jansen apply the principles of a case study to demonstrate the practicality of their work, they do not conduct an empirical study which is different to our study.

Falessi et al. picked up the idea of capturing the rationale for design decisions (also known as design rationale documentation (DRD)) in 2013 and investigated what information items are likely to be required for executing an activity in working on a software system [12]. This is because they state that documenting everything is inadequately inefficient and too onerous to be used in industrial software development. Therefore, they suggest to reduce the amount of

documentation through a value based customization. Meaning that only those rationales are captured that are likely to be used in future activities. They hence conducted two controlled experiments including 75 master students of computing science. Each student had to perform an activity on a system using multiple information including DRD for the specific system. They then had to categorize these information as useless, required, or optional. Their findings suggest that customizing the DRD documentation can reduce the amount of documentation that needs to be done by 50%. They hope that their findings can motivate software practitioners in documenting the customized crucial design decision rationales required and supporting future activities in their software systems. This study can have an beneficial but also disadvantageous effect on our study. One the one hand, it may motivate open source software developers to put effort in documenting their design decisions and on the other hand, lead to a customization of the DRD that prevents of documenting the rationales for incurring ASs.

In 2015, Groher and Weinreich conducted a survey on architectural decision making processes in industrial software development [18]. They found out that decision making processes are influenced mostly by organizational (team size/organisation), individual (experience, education, passion), cultural (degree of freedom), business (domain, costs, risks, time-to-market), and project (duration, kind of project) factors. Surprisingly, domain and company size may not have such a great influence on architectural decisions making than one may expect. In addition, they found out that most teams work in an agile environment where developers w.r.t their responsibilities are mostly free to make decisions. However, they also found evidence that usually either more people are involved in high-impact decision making situations or people with designated roles are consulted. These findings have been derived by interviewing 25 experts from industry. This study is related to our study because it provides information what factors influence decision making. We can pick up those factors and include them in our case study to find the rationales for incurring ASs. The study of Groher and Weinreich however, differs from our study since they investigate the general area of decision making processes in software development but not in incurring ASs in particular. In addition, their method is based on interviews where our study focuses on analysing documentation artifacts related to versions that incur smells.

Shahbazian et al. adapted the aforementioned findings on ADDs in 2018 in order to recover undocumented decisions from issue tracking systems [38]. They agreed that rational and consequences are important parts of the decisions and refined the decision category to simple,

compound, and crosscutting decisions. With utilizing and combining information derived form GitHub and Jira they created a technique of recovering design decisions. This technique was applied to two highly complex software systems (*Hadoop* and *Struts*) in an empirical study. They showed the possibility of preserving design decisions form history information such as Git commits and Jira issues.

Our research follows the presented literature in that way that we use the findings of [7, 21, 22] concerning the relation of ADDs and the rationale for this decision. We further use a similar approach to extract the design decisions form existing code and issue repositories as in [38]. We also conduct an empirical analysis on open software projects. However, the main contribution of our work is to find the rationales for ASs and not to recover ADDs in general. Also our approach differs in the extraction mechanism of the decisions. We apply existing AS detection tools in order to gather versions where one or more smells are incurred in the system. We than map these version this smell occurs to combine it with the Jira issue to extract the rationale. At this point we use the principle of [21] that a decision can lead to another decision. With this we are able to extract a rationale for change decision and for the smell itself.

3.3 RESEARCH ON SELF-ADMITTED TECHNICAL DEBT

The *Technical Debt* (TD) metaphor was first used 1992 by Ward Cunningham in order to describe the situation where immature code is used in order to meet a deadline or to deliver software quicker to the market [11]. Cunningham described that even if the software is functioning well, developers may have gone into debt. He argued that a little debt can be tolerated when it allows gaining value such as speeding up development. Nonetheless, it becomes dangerous when the debt is not paid back. Further, it can bring the entire software engineering process to a stand-still in case of a high debt load.

While other researchers focused on the implications and management of TD Potdar and Shihab examined the impact of technical debts that are intentionally introduced into software called Self-Admitted Technical Debt (SATD) [31]. They analysed the source code of four large open source software projects in order to quantifying the amount of SATD, the reasons behind the intended admission, and whether the debts are removed after their introduction. The results suggested that 2.4 - 31.0% of the files in the project contained SATD. Furthermore, time pressure or complexity seem to be no reason for deliberately introducing TDs. However, it appears that more experienced developers tend to add more SATD. Finally, only 26.3 - 63.5% of the intentionally

introduced debts had been removed afterwards.

In 2015, Maldonado and Shihab focused on the different types of TD introduced by SATD [34]. They conducted an empirical analysis on 33K comments derived from five open source software projects in order to determine different types of TD. They found five different types of debts, namely design debt, defect debt, documentation debt, requirement debt, and test debt. Maldonado and Shihab further discovered that the majority of the SATD introduced in the analysed systems are design debts. The second most frequent type was requirement debts. The remaining types occurred in a low frequency considering that they represent less than 10% of the debts they found.

The contributions of our work follow a similar approach as Potdar and Shihab. We are also trying to find out how much of the ASs are self admitted. As the results of Maldonado and Shihab indicated that no architectural debts are added intentionally, this would be a first proof that ASs are actually a sub-category of SATD s. In addition, we try to find the rationales for intentionally adding ASs similar to finding the rationales for SATD. However, the scope of our study is different. We are focusing only on ASs but not on TD in general as it is done in [31]. Furthermore, our detection approach is a different one since we apply existing smell detection methods and tools in order to connect versions of software systems to issues in the corresponding tracking tools. The presented related work on SATD parses code comments to detect the debt instances. Finally, our study also aims to cover the rationales for inadvertently introduced architectural smells.

4

CASE STUDY DESIGN

This study is about the rationale developers make when inducing ASs into their software system. In this section we describe our study design that will help us to derive this rationale. We therefore first provide information on the principles and theoretical basis according to which this study is designed (Section 4.1). We then phrase the goal of our study and formulate the corresponding research questions in Section 4.2. Subsequently, Section 4.3 depicts how the projects that have been used were selected. Finally, Section 4.4 concludes with describing the processes necessary to derive the data that is required to answer our questions.

4.1 RATIONALE FOR CASE STUDY DESIGN

In order to find the rationales for incurring ASs into a software system we conducted a case study. A case study "is an empirical inquiry that investigates a contemporary phenomenon within its real-life context" [43]. As ASs are such a contemporary phenomenon we decided to design our study following the guidelines for "Case Study Research in Software Engineering" described in Runeson et al. [33].

A case study allows a study design that includes a high degree of realism, allows a flexible study setup, and tolerates intentional subject selection [33]. These characteristics are beneficial because real world examples will indicate faulty behavior during software development. In addition, the flexibility of case studies supports an iterative study design where the setup of the study can be aligned to new findings encountered during the collection of data. Finally, selecting the study subjects intentionally allows to study those software projects where (1) information about the project is accessible and (2) only subjects can be selected where the documentation quality meets the requirements to answer the research questions of our study.

As far as our knowledge goes, no other case study has been conducted yet with the objective to investigate the reasons for incurring ASs. Several hypotheses have been proposed why software practitioners add smells into a system as mentioned before. However, we could not find any evidence and hence have no theoretical background that explains why ASs are added to a system. Therefore, our case study design has to be inductive as explained in [33]. More in detail, this case study will result in observations that may reveal patterns on

which we can define tentative hypotheses, i.e. if we find a certain characteristic in the evolution of architectural smells we can apply this in our study design. In the best case, we are then able to define a theory, based on our empirical research, that provides an explanation for incurring new AS instances into software.

The process of a case study is divided into the *case study design*, *preparation for data collection*, *collecting evidence*, *analysis of collected data*, and *reporting* [33]. The *case study design* will define, among others, the goals of this study and provide details about the cases and units of the study. Elements that belong to the case study design such as rationale and purpose of this study have been already provided in our introduction in Section 1. Similarly, theoretical background information and related work have been discussed in Section 2 and Section 3, respectively. The goal and research questions will be presented in Section 4.2. A description of the case selection and units of analysis is explained in Section 4.3.

The *preparation for data collection* defines the procedures and protocols for data collection [8]. This step describes how the data sources have been selected for this study. An important aspect is that the flexible design approach of case studies allows an iterative refinement of the data collection. This way one can report how raw data was filtered and what characteristics have been found during the analysis of the data sources that have been transpired to be useful. In addition, this step also defines how the resulting data is organized. This will help to increase the quality of the study because it ensures that no data is lost due to disorganisation. The description of the data collection is presented in Section 4.4

Subsequently, the step of *collecting evidence* describes how the data used in this study was collected. This includes for example the tooling that was applied to the raw data derived from the data sources. In addition, it describes how the data is stored and how someone can access the data in order to provide transparency of our research process. For this study, this is the detection of ASs and the extraction of data the presumably entails the rationale for incurring the smells.
Finally, the *reporting* of the findings and the *analysis of the collected data* presents the findings and describes how the data is analysed and interpreted. For our study, these aspects are discussed in Section 5 and Section 6.

4.2 GOAL AND RESEARCH QUESTIONS

The objective of this study is to achieve knowledge on the rationale for ASs. As far as we know this study is the first that focuses on this

topic. We apply the Goal-Question-Metric (GQM) approach [4] in order to specify our goal.

The GQM helps to specify goals for a project in a purposeful way [4]. It establishes a framework that not only allows the interpretation of data with respect to the stated objective but also to trace the way from the objective description to the data. In general, it defines a goal that encompasses its purpose, the issue it is defined for, objects that are used to measure it, and which point of view it takes. The goal is then specified by various questions characterizing the assessment of achieving the goal. Finally, metrics define how the data derived from the object of measurement is associated with every of the aforementioned questions. Thus, the metrics allow to answer the questions in a quantitative way. The goal formulation for this study is:

> *Analyse the rationale for incurring architectural smells using issue and commit information related to a software version which incurs a new architectural smell instance from the point of view of software engineers in the context of Apache open source projects mostly written in Java with the purpose to find the reasons for incurring architectural smell in software.*

In general, all research questions of this study specify the objective of our study. Therefore, each question focuses on a different aspect of the rationale for ASs. The questions are presented below and we five here a brief overview on the focus each question has. ***RQ1*** focuses on the evolution of architectural smells. ***RQ2*** focuses on the issue types that motivate software developers to incur ASs w.r.t the three instability architectural smell types mentioned in Section 2.3. ***RQ3*** examines whether the prioritization of issues can indicate whether architectural smells are incurred. Next, ***RQ4*** seeks to find how different developer constellations resolving issues influence the introduction of smells. Lastly, ***RQ5*** studies the decisions, made by the developers and captured in their discussions on code or issue repositories, for adding a ASs to the system during development. The research questions are answered by dividing them into several sub-questions which further specify the aspect the particular question focuses on.

RQ1 What are the characteristics of the evolution of architectural smells?

a) How do architectural smells evolve across software project lifetime?

b) How big do architectural smells evolve?

c) How long does it take architectural smells to evolve?

The research on the evolution of architectural smells has just begun. However, only understanding the evolution of architectural smells enables us to determine when a new smell actually is incurred into the system. This is important for this study since the main goal is to understand the reasons why smells are incurred into software. The most important information to answer this question comes from the architectural smell itself. It composes by all variations of it that are involved in its evolutionary changes over time. Therefore, the first metric that we define is the total number of smells newly incurred in the project. Another metric that helps us understanding the evolution of smells consider the number of variations per smell and the corresponding time-span in which the variations are added. Other metrics that we use for this research question are the number of smell expansions and the number of splitting points. Additionally, we use several information about the size of the smell variations measured by the number of components involved in the particular information. All these metrics enable us to create commonly known statistical measurements that provide information on how smells evolve during their life-spans.

RQ2 What are the issue types motivating developers to incur Architectural Smells?

a) How many smells are incurred by the different issue types?

b) How is the distribution of issue types introducing new smell instances?

The aim of this research question is to find the motivation for incurring architectural smells. An answer may indicate specific situations in which it is more likely that a new smell instance is added to the architecture. This information can be used in the management of software development. Engineers can take precautions to prevent polluting the system unnecessarily with ASs when they work on issue types that are more likely to incur smells. The metrics used to answer this research questions are derived form the Jira issue and the architectural smells themselves. Every smell is aligned to a specific version of the Git commit history. This version is usually related to a Jira issue which determines the issue type. Additionally, the smell tells us about its smell type. The first metric used to extract information from these attributes is the number of smells that are incurred by each issue type and the ratio of it. The second metrics accumulates the number of issue types that incur a specific smell type or combinations of smell types. In order to compare these data the second metric is normalized by the total number of issues for the corresponding project.

RQ3 How does the prioritization of issues influence incurring architectural smells?

a) How many smells are incurred by which issue priority?

b) What are the priority levels of issues which trigger architectural smells?

This research question investigates how software practitioners prioritize issues that incur smells. Answering this question can reveal whether the categorization made by developers is adequate with regards to architectural design flaws. If for example the bulk of smells are incurred in the priority category “Trivial" or “Minor", one can see this as an indication that developers are not aware about development situations that result in architectural significant changes. This information may be helpful to guide software teams in their prioritization of development tasks. The metrics that can be use for this question are similar than those defined for ***RQ2***. However, they replace all issue type information with those of issue priority which results in number of smells incurred by each issue priority and the number of issues ordered by priority that incur a specific smell type or combination of these types.

RQ4 How does the alignment of developers impact introducing architectural smells?

a) What is the experience of developers, who commit architectural smells?

b) How many developers participate in incurring architectural smells?

Another important aspect of why smells are incurred into the system is the choice of developers. Therefore, ***RQ3*** focuses on the what kind of developers and how many work on the system changes that introduce new ASs. One may assume that an experienced developer is more aware of the problems arising from a degradation in inner system quality and hence avoids incurring ASs during development. Additionally, one may also agree that more developers working on the same issue combine their knowledge and preserve commonly accepted design principles. An answer to these research questions can thereby guide development teams in personnel decisions. If for example, inexperienced developers add more smells to a system then experienced once and a group of developers working on an issue together incurs less smells than a single developer. Then, a solution to prevent adding new smells may be to group experienced and inexperienced developers and avoid developers working alone on an issue. The metrics defined for this question are number

versions that incur architectural smells by a single developer and number of versions incurring architectural smells by two or more developers. Furthermore, the number of versions with architectural smell that a specific developer incurred.

RQ5 What are the reasons mentioned in the documentation for incurring an AS instance?

a) How many AS are admitted deliberately?

b) What are trade-offs made by incurring AS?

RQ5 focuses on the rationale for the decision that is made for incurring ASs to the system. The answer to this question may be - according to our knowledge - the first proof or indication of whether developers incur ASs deliberately to achieve a certain goal as often declared in various literature. Further, when this answer is positive, what are the motives also known as trade-offs for doing this. If the findings suggests that ASs are only added inadvertently, the software engineering community is in need to find ways to prevent this from happening. In case the findings suggest that developers pollute the system deliberately, these situations need to be studied in order to determine the value that is gained form it. This may create a trade-off catalog towards other system qualities at the expense of inner system qualities such as maintainability and so on. Such a catalog would be a valuable tool in technical debt management. The metrics used to answer this research question are of qualitative nature. We use *Issue Context*, *Issue Information*, *Smell Type(s)*, a description of how the smell was newly incurred into the system, *Reason for approval*, and *Rationale for smell* in order to provide all information required to understand why an architectural smell was incurred into the software project.

4.3 CASE SELECTION

This case study is designed as a single embedded case study with several units of analysis following the categorization for case studies in software engineering by [33]. We see the open source projects of the ASF as the context of this case study. Therefore, we cannot claim that our findings have a general applicability. However, they can indicate certain behavior of incurring architectural smells into a software system. This may point future research into a certain direction that investigates our findings in a different context, i.e. commercial software projects etc.

The case that we are studying is, as already mentioned, the rationale for incurring ASs into open source projects. For this we analyse the

smells of six different open source projects under the supervision of the ASF. Hence, all smells of one project take the role of a unit of analysis. We further subdivide these units into different types of corresponding documentation artifacts. These artifacts are related to the particular software version that incurs a new smell instance. Those are namely, **version commits** that encapsulates the changes that have been made for this version, **Jira issues** related to the version commits, and malicious documentation artifacts that are attached to either commits or issues. Furthermore, all attributes belonging to an commit or issue such as ids, comments, or assigned developer are part of these sub-units as well.

Apache was chosen because it is the largest open source foundation in the world according to their home page[1]. In addition, the tools used in this work (see more details below) require Java projects and the ASF oversees more than 200 projects of this programming language. Furthermore, Apache projects are also used in other software engineering studies because of their well maintained code and issue repositories (e.g. in [39]). Table 8 provides a list of all projects completely analysed in this study.

We defined the process of selecting the projects in Protocol 1. We first, provide an overview on this protocol and later describe each step in detail:

Protocol 1:

1. **Project Selection** - select all Java Projects of the ASF
2. **Categorize Project** - align each project to one of the aforementioned categories
3. **Extract Comment Information** - Request the number of comments for each Jira issue of each project. This step was performed via an automated python script using *Python Jira API*[2] and results where stored in a spreadsheet for further analysis.
4. **Ensure adequate project size** - Discard projects with less than 1000 issues.
5. **Rank projects** - determine rank of remaining projects using these metrics:
   a) Calculate percentage of commented issues
   b) Calculate the average of all commented issues
   c) Normalize both metrics using *min-max*-normalization

1 https://www.apache.org/ - accessed 01.07.2020
2 https://jira.readthedocs.io/

   d) Sum both normalized metrics and sort by highest value

6. **Evaluate comment quality** - Manually determine the quality of the comments by random sampling from the top ranked projects until 5 projects for each category have been found.

Step 1: Project Selection

All systems that are analysed in this case study need to fulfill certain requirements. First, as already mentioned, they need to be written mostly in Java. Second, the projects issue have to be managed on Jira and publicly accessible. Finally, the code repository of each system must be a managed by the version control system Git and hosted as a public project on GitHub. The decision for Jira stems from two main reasons. First, Jira in combination with ASF allows open access to the information we require to answer our research questions. Second, Jira has a general structure that supports us with creating protocols for extracting the information we require in the same way for all projects we analyse. Additionally, the Jira API allows a (partial) automation of executing the processes derived by the protocols. A similar reasoning applies for Git and GitHub.

Step 2: Categorize Projects

However, these requirements are almost fulfilled by all 237 Java projects of the ASF and analysing all of them is out of scope of this study. Therefore, we present a filter process that led to the six analysed projects. In order to get a broad applicability of the results, the projects selected for this study must belong to different software domains. After examination, we manually divided the projects into six categories because the categorization of the ASF is in some cases too specific, indefinite, or misleading. We categorize the projects in order to ensure that the projects we analyse belong to different domains, to ensure a broader applicability of our findings. Our categories entail *data store systems*, *middleware systems*, *frameworks*, *analytic engines*, *libraries*, and *web management*. We understand *data store systems* as those system concerned with persisting data such as *database management systems*, *data warehouse systems*, or support tooling for those systems. The *middleware* category describes software systems that provide communication among several applications such as messaging systems [19]. A *framework* is an integrated set of components that collaborate to produce a reusable architecture for a family of applications [28]. Under *analytic engine* we categorize those software applications that are used for analysing large amounts of data. A library is a “controlled collection of software and related documentation designed to aid in software development, use, or maintenance" [20]. More in detail, this

collection can be used to reuse certain behavior within a software program without the need of the development team to implement this behaviour themselves. The last category composes by software applications that can be used for web management such as search engine development and alike.

Step 3 & 4: Extract Comment Information & Ensure Adequate Project Size

As discussed later in this section, a crucial part of deriving the rationale for incurring ASs is analysing the discussions of the developers within issues. We found that not in all *Apache* projects those discussions are documented in Jira. Therefore, we selected projects where the amount of comments and the quality of the comment content are promising to capture the rationale for ADDs. Accordingly, we automatically analysed the comments for each issue and calculated the ratio of commented issues and the average number of comments per issue for each project. We further discarded all projects with less then 1000 issues in order to guarantee that the system's architecture has evolved enough to consists of ASs.

Step 5 & 6: Rank Project & Evaluate Comment Quality

After applying *min-max normalization* to the aforementioned comment metrics of the remaining projects, we achieved an ordered list of all projects sorted by category. For those projects we manually evaluated the quality of the comments by taking random samples beginning with the best rated projects until we found five projects for each category. A list of all 25 projects can be found in Table 1 to 5[3]. Unfortunately, we had to discard more projects due to errors in the execution of our analysis pipeline which is discussed later.

Project	Number of Issues	Number of Comments	Issues with comments	Commented issues in %	Avg. comments in commented issues
Derby	7,059	60,781	6,731	96	9.03
Cassandra	15,493	105,146	14,445	94	7.27
Tajo	2,182	16,165	1,915	88	8.44
Directory Studio	1,231	3,857	1,102	90	3.5
Accumulo	4,745	22,728	3,733	79	6.08

Table 1: Projects for Data Store Systems

3 A list of all projects and their corresponding normalized values can be found on https://github.com/ThorstenRangnau/Jira-Project-Analyzer

Project	Number of Issues	Number of Comments	Issues with comments	Commented issues in %	Avg. comments in commented issues
MINA	1,083	4,698	1,007	93	4.66
Sqoop	3,130	15,802	2,579	83	6.12
Axis2	5,877	17,574	5,393	92	3.25
ActiveMQ	7,140	22,475	6,459	91	3.47
Camel	14,570	46,830	11,618	80	4.03

Table 2: Projects for Middleware

Project	Number of Issues	Number of Comments	Issues with comments	Commented issues in %	Avg. comments in commented issues
ZooKeeper	3688	35457	3205	87	11.06
Mani-foldCF	1635	9581	1584	97	6.04
Bigtop	3313	18196	3018	92	6.02
OFBiz	11348	52625	10715	95	4.91
Cocoon	2362	8284	2158	92	3.83

Table 3: Projects for Frameworks

Project	Number of Issues	Number of Comments	Issues with comments	Commented issues in %	Avg. comments in commented issues
Hadoop YARN	9888	129867	8954	91	14.50
Hadoop HDFS	14419	176055	13444	94	13.09
Hadoop Common	14662	157523	13863	95	11.36
Hadoop Map/ Reduce	6941	68484	6320	92	10.83
Phoenix	5716	43205	4936	87	8.75
Flink	16283	114222	14238	89	8.02

Table 4: Projects for Analytic Engines for Big Data Processing

Project	Number of Issues	Number of Comments	Issues with comments	Commented issues in %	Avg. comments in commented issues
PDFBox	4775	36027	4700	99	7.66
Log4j 2	2781	17179	2492	90	6.89
Commons Math	1505	7816	1457	97	5.36
Tika	3041	16612	2808	93	5.91
Commons Lang	1481	7187	1406	95	5.11

Table 5: Projects for Libraries

4.4 DATA COLLECTION

This section provides information about the collection of data required to answer the research questions defined above. We divide the data collection into three steps: pre-analysis, quantitative analysis, and qualitative analysis. The pre-analysis part concerns all activities required to identify the versions in which a new smell is incurred in the system and the mapping to the corresponding Jira issues. These activities are described in Section 4.4.1. The quantitative analysis (Section 4.4.2) concerns all steps necessary to process and aggregate the data from GitHub and JiraḞinally, Section 4.4.3 describes the steps taken in order to analyse the data qualitatively. Especially, the two latter steps include a detailed description of how the data is aligned to the metrics defined in Section 4.2. These metrics are important in order to answer the research questions.

Before we start and explain the processes to derive the data being used in this study, we want to provide some information about the data that we collect as well as the underlying methods of the collection.

Usually, case studies tend to use qualitative data which are considered to be 'richer' [33]. It can however be beneficial to use both quantitative and qualitative data in a case study because the quantitative data provide a better understanding of the studied phenomenon. In this study we use both types of data which is also known as *mixed method case study*.

Runeson et al. describes three *degrees* of data collecting techniques which determine what kind of data source is being used in the case study [33]. The first degree concerns methods extracting information from the data source directly such as interviews or observations. The second degree is when one collects data indirect form a direct source (video of an interview, monitoring developer working with software engineering tool, etc.) In this study we use the third degree of data source. We collect our data independently from the information source via work artifacts. More precise, the developers (direct data source) document their work in a Jira issue or a Git commit which we then analyse. We chose to use third degree data sources because it allows us to independently design our case study from the data source[4].

The entire research process that results in the data that is used to answer the research questions is depicted in Figure 12. As described

4 A wise decision as we encountered during the error prone process of the smell detection.

above this process is divided into three major steps: pre-analysis, qualitative analysis, and quantitative analysis. We will describe all three steps in detail in the subsequent sections.

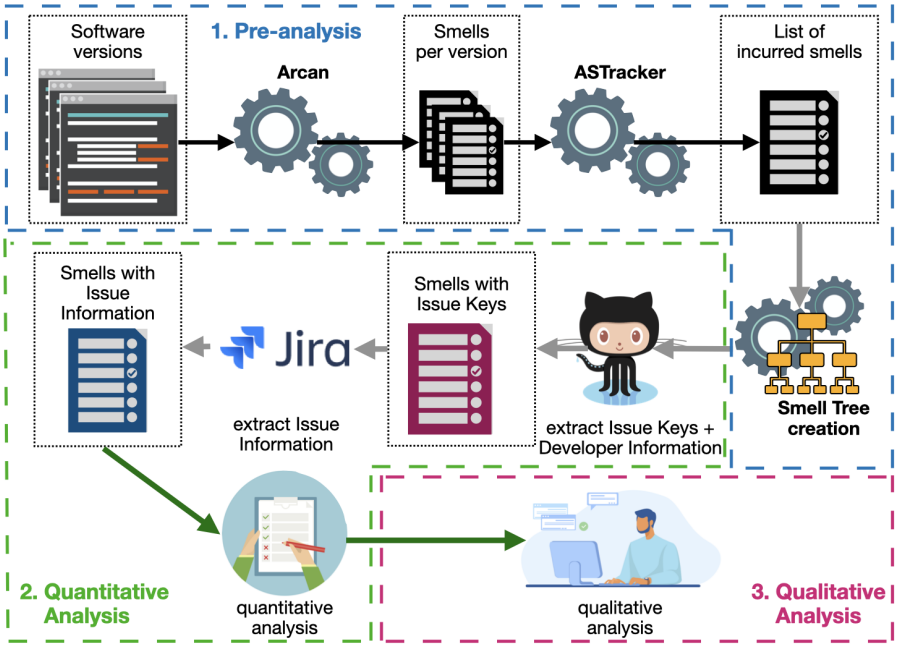


Figure 12: Overview on the research process of this case study

4.4.1 *Pre-Analysis*

As Figure 12 indicates, the pre-analysis composes of three processing steps: Arcan analysis, ASTracker analysis, and smell tree creation. In this section we discuss them individually. All processes of the pre-analysis phase are defined by the following protocol.

Protocol 2:

1. **Arcan Analysis** - start Arcan analysis
   a) Clone Git project to disc
   b) Checkout first commit
   c) Start Arcan configured to scan every version
   d) Check analysed time-span of project and discard project if too less versions are scanned
   e) Check logs for errors and missed packages and discard project if errors are to severe or source code packages are excluded although part of the system
2. **Filter Arcan Files** - filter versions not included in default branch
3. **Detect Version gaps** - Determine the gaps for each version

4. **ASTracker Analysis** - start ASTracker analysis of filtered Arcan files

5. **Create smell trees** - align Algorithm 1 for cyclic dependencies, unstable dependencies, or hub-like dependencies and store smell trees to disc in a csv file

Step 1: Arcan Analysis

For the smell detection we used Arcan (in combination with ASTracker) and Designite to varify the existence of the smells. We chose to apply Arcan within the analysis pipeline because it provides automatic scanning of each version of a software project which is managed by Git.

As indicated, Arcan takes all software versions of a project as input and creates a graph-file for each version as output. At first, Arcan creates a dependency graph which includes all dependencies among packages [13]. This graph is then used in order to detect the smells that are present in the system. Hence, all information about the smells of a specific version are included in the corresponding graph-file. Eventually, the Arcan analysis results in all smells per version of a software system.

Problems with Arcan Analysis

We encountered several problems with Arcan. In order to give an overview we first present a list of all these problems. Subsequently, we describe the indications of these problems, e.g. projects that we had to discard because of this problem.

1. **Long run-time** - The first problem with using Arcan was that its functionality for analysing all versions of a software project was assumed to skip several versions and only scan every 100 versions of the commit history in Git. This is however not applicable in our study since this raises the risk of missing those versions that are actually incurred in one of the skipped versions. Although, Arcan is capable of detecting the smells incurred in these gaps, it aligns them to the wrong commit id. This of course biases our results and has to be fixed. Nonetheless, we were able to configure Arcan such that no versions were skipped (which did not work 100% and is discussed below). Scanning every version however proved to increase the execution time of a single project exorbitantly (for example *ActiveMQ* has 10,630 versions). Since execution was not applicable to a single laptop machine, we switched the execution of Arcan to the *Peregrine HPC Clus-*

ter[5], the high performance computation cluster of the University of Groningen.

2. **Sequential execution** - Unfortunately, during the development of Arcan, scalability and/or parallel processing presumably have not been addressed at all. Therefore, we have not been able to parallelize the execution of Arcan in order to speed up the performance. However, this is important to achieve an acceptable run-time during scanning all versions of a software project. Even on the HPC cluster we were not able to analyse all 30 projects entirely due to exeeding resource limits, namely allowed computation time (max. 240 h) or available memory. Luckily, Arcan stores all created graph files to disc once they were created which enabled us to partially analyse several projects. This is acceptable for this study since we can still extract useful information of a project even if only half of its versions are analysed. However, we had to discard some projects here because even applying the maximum run-time of 10 days and a maximum memory allocation of 128GB resulted in analysing less of the first year of the project. These projects are: *Accumolo*, *Camel*, and *Log4J2*.

3. **Excluding source code** - Another crucial problem of this first part of the pre-analyse phase was that Arcan excluded several source files of a project. A closer analysis of this problem revealed that the problem was in the software project itself but not in Arcan. This is because the developers of those projects did not follow the package structure convention recommended by Maven[6]. Nonetheless, it is possible to configure Arcan such that it receives a list of directory path in which it will search for source code. We deemed this however as out of scope of this thesis because this requires manually detecting all packages that include source code of the entire project. Especially the fact that the package structure most likely changes as the projects evolves over time requires to find all packages in all versions. Eventually, we decided to discard those projects from the analysis. Those projects where *Derby*, *OFBiz*, and *ManifoldCF*.

4. **Problems with Git API** - Furthermore, Arcan appears to have sometimes a problem with checking out the next version/commit, as we encountered several exceptions for this in the log files. Because of this Arcan ignores these versions in the analysis. Although technically not failing, this provided us with the problem that Arcan detected smells that may be incurred in those versions are aligned to the wrong version which would

5 https://wiki.hpc.rug.nl/start

6 https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

again bias our results. This error is presumably due to the *Java* Git API used by Arcan which seems to have difficulties with checking out a version when the changes between this version and the current version are to big. However, we were able to mitigate this problem to a minimum with checking out the first commit of the project before starting the Arcan analysis. However, there was one project where this problem still remained probably due to a greater refactoring of the package structure. This project was *ZooKeeper* which we also discarded.

5. **Scanning wrong versions** - We also discovered that Arcan analysed sometimes versions of a software project that are not part of the default branch (master or trunk). Unfortunately, each software project seems to have its own branching strategy and it is hard to determine which branch is the one that is used for releases. Nonetheless, we follow in this study the strategy to only include those versions that are committed to the default branch. Therefore, we filtered all non-master commits out of the analysis (**Step 2**) Since, we were not able to determine why Arcan included non-default branches into the analysis we discarded all graph files that were created for these non-default branch versions.

6. **Skipping versions** - We further observed that Arcan skipped several commits that are actually included in the master branch. This phenomenon is partially due to the master branch filtering that we mentioned before, however we also found this without the filtering. Our workaround for this was to find these gaps in the commit history (**Step 3**). This allows us subsequently to either discard smells that are identified in a version that follows such a gap or to manually determine whether the smell was incurred in this version or find the correct version. We further describe this process in Section 4.4.2.1. In addition, the number of gaps that we found are displayed in Table 9.

7. **Miscellaneous errors** - Finally, there were some other problems that we were not be able to mitigate in the Arcan analysis. For some projects the analysis with Arcan failed due to internal exceptions during the execution. We discarded these projects because even if we could technically process the graph files from Arcan we decided to not bias our results due to unknown problems. Those projects were, *Flink*, *Axis2*, and *BigTop*.

Step 4: ASTracker Analysis

The second step of the pre-analysis phase is the ASTracker analysis. Here, ASTracker takes all graph-files created by Arcan as input and creates a list of all incurred smells over the life-time of the project [36].

Therefore, the graph-files are scanned chronologically and the smells that are present in Version 1 are mapped to the smells in Version 2. Afterwards, the same happens with the smells of Version 2 and Version 3, and so on. ASTracker maps the smells by their similarity. This is done by applying the *Jaccard similarity index* which determines the percentage of components that are included by a smell in the current version and in the smells of the previous version. The components are identified by their full package name. This way one can track the evolution of a single smell through the entire history of the system. The results of this mapping are then stored in a csv-file. In this file, the smells are ordered by the smells they belong to. In addition, for each smell one can see in which version this smell was detected first.

Problems with ASTracker Analysis

Unfortunately, we also encountered several problems with applying ASTracker. Again, we first list them and describe them subsequently in detail.

1. **Long run-time** - At the beginning of our analysis we encountered a similar problem regarding memory usage and performance while executing ASTracker as we did with Arcan. As well as Arcan, ASTracker was originally designed to analyse every 100th version of a software project. Processing all graph files generated by Arcan overstrained ASTracker and its execution exceeded the available resources even for smaller projects. Fortunately, we were in contact with the developer of this tool, who supported us with a new, better performing version. However, we still ran into problems with executing those projects with many versions and very large graph files (Cassandra: around 25,000 files with around 10MB each). Here we needed to use high memory machines that provided more than 128GB RAM.

2. **Wrong versions** - After we were able to execute ASTracker and evaluated the results, we could not always find evidence for smells that ASTracker marked as new for a certain version. Very obvious wrong results claimed that adding descriptions in a single documentation file should have resulted in several new cyclic and unstable dependencies. Yet, evaluating the smells being present in a certain version with the tool Designite revealed that around 90.9% of all smells are actually present in the system in this version. We accept the missing 9% due to different detection algorithm or divergent configurations by the two tools [3]. However, this does not explain why a text-file change incurs multiple ASs.

3. **Imprecise similarity mapping** - Further investigation revealed that the version we use for ASTracker only tracks a smell if it

is able to find the very same smell in the consecutive version. If it is not present there but re-appears in the following version it is regarded as a new smell. Additionally, we discovered that the *Jaccard similarity index* may not fit very well for linking smells together. This latter discovery may also be an explanation why certain smells are "incurred" out of the sudden in a version without any clue how it could be added in the diffs of the corresponding Git commit. We created two workarounds to mitigate this problem: removing of redundant smells and a heuristic for linking smells together.

Step 5: Smell tree creation

Since the goal of this study is to identify the rationale for incurring instability ASs, we need to find these versions where a new smell is added to the system. In order to find those we first take only those smells and their corresponding versions of the ASTracker analysis which indicate that this smell is newly incurred into the system. We then compare all components of all smells and sort out redundant smells without regarding the order of the smells. We keep the earliest occurrence of this smell. This way we ensure that we really receive the version in which the smell is incurred.

However, even with filtering the duplicated smells we still receive smells where we could not find evidence that these smells are indeed incurred in the version ASTracker indicates. A closer look at the remaining smell list revealed that these smells may be still related to one another. We support this assumption since even after filtering the duplicated smell instances, the number of smells supposably incurred in one version is very high. This lead to the assumption that not all smells are yet properly aligned to one another.

As mentioned earlier we believe that the mapping strategy of ASTracker may not be efficient for our work as we require. Also we mentioned that ASTracker was originally not created for such a fine-grained analysis as it is conducted in this study. This may be the reason why not all smells are accurately connected to each other. We mitigated this problem with a simple heuristic that maps similar smells together and thereby creates a tree-like structure that represents the smell evolution of a single smell over time. Algorithm 1 provides an

overview on our approach.

```
input  : Set S containing all smell variations of one smell
         type sorted by component name
output : Set ST containing all smell trees
1  begin
2      // align smell variations starting with the same two
        components
3      SV ← AlignSmellVariations(S)
4      for i ← 0 to len(SV) − 1 do
5          // get a Set of all smell variations of one smell
6          smell ← SortByDate(SV_i)
7          // set oldest smell variation as root of smell tree
8          ST_i ← SetRoot(smell_0)
9          for j ← 1 to len(smell) − 1 do
10             if TreeHasOnlyRoot(ST_i) then
11                 // align node to root if tree only has a root
                    node
12                 ST_i ← AlignNode(smell_j, root)
13             else
14                 // create a list for all m1 and m2 values
15                 m1, m2
16                 for k ← 0 to len(ST_i) − 1 do
17                     // compute m1 and m2 for each
18                     m1_k ← ComputeM1(smell_j, ST_ik)
19                     m2_k ← ComputeM2(smell_j, ST_ik)
20                 end
21                 parent ← FindParentNode(m1, m2)
22                 ST_i ← AlignNode(smell_i, parent)
23             end
24         end
25     end
26 end
```

Algorithmus 1 : Create smell trees for all smells of one smell type

This heuristic takes a set of all smell variations for one smell type. It will align the smell variations based on the components. We receive the components involved in a smell variation from ASTracker. Unfortunately, we receive no information on the role a component has in the smell. For example, if the smell is star shaped cycle we do not know if the first component of the component list is the center component or a component that forms a beam. The same counts for the unstable dependencies and hub-like dependencies. Because of this we leave the order of the components as it is. We simply align all smell variations to the same smell where the first two components

are the same. This leaves of course the risk of aligning the wrong components together since we have no idea what the role of the components is. Nonetheless, our experiments with this heuristic resulted in promising smell roots which indicates that we indeed found the very first variation of smell that is incurred in the designated version.

After aligning all smell variations to one smell we find the root node by sorting the variations by date and take the oldest. Then we iterate over all remaining smell variations and align them to the most similar tree node. In the first iteration one obviously needs to align the smell variation to the root of the tree because it is the only node being present in the tree at this moment. Note that they are already sorted by date so that aligning a new node to the tree will automatically create a tree where each parent node is older than the child node.

In order to align the remaining smell variations we use two metrics we call *component difference* (m1) and *component coverage*(m2) in order to determine the best fitting parent node. To find the best parent node one has to calculate m1 and m2 for each the node in the tree. Both metrics are calculated using information about the components of the smell variation that has to be added to the tree and the tree node it is compared with.

The m1 metric determines the difference in the number of components. It is depicted in Algorithm 2. We found out by creating the smell trees manually that a smell mostly grows or shrinks. In addition, we found that growing or shrinking rate is usually only marginal (one or two components). Therefore we not only calculate the absolute value of the difference of two component numbers but also benefit the growing or shrinking rate of a single component by subtracting one from the aforementioned difference.

One additional reason for this is that we found out that a smell variation with the component length seldom fits as a child node to a smell variation with the same component length[7] As a result for this metric one can say that the minimum of all m1 values calculated for a smell

7 One exception is when only the root node is present in the tree but there we do not use the metrics at all but simply add the variation as child node to the root node.

variation determines the node where the current smell variation evolutionary fits the best.

```
input 1 : Set c1 with the components of the current smell
          variation
input 2 : Set c2 with the components of the current smell
          node of the tree
output  : m1 difference of component numbers with
          benefit for growing/shrinking by 1

1 begin
2     // benefit growing/shrinking of three with subtracting
       1
3     diff ← len(c1) − len(c2) − 1
4     // make value positive to allow comparison of
       shrinking
5     m1 ← |diff|
6 end
```

Algorithmus 2 : Heuristic to calculate m1 metric: Function `ComputeM1` in Algorithm 1

Although m1 determines the best evolutionary development of a smell tree, it only considers the number of components involved in a smell variation but not the components themselves. For this reason, the m1 value for comparing the two smell variations SV_1 with components (A, B, C, D, E, F, G, H) and SV_2 with components (A, B, I, J, K, L, M, N, O) although SV_3 may fit better to the smell variation SV_3 with the components (A, B, I, J, K, L, M) simply because SV_3 has not enough components considering only the m1 metric.

```
input 1 : Set c1 with the components of the current smell
          variation
input 2 : Set c2 with the components of the current smell
          node of the tree
output  : m2 the coverage of c2 components over c1
          components in percentage

1 begin
2     // calculate how many components of c2 are in c1
3     coverage ← CalculateCoverage(c1, c2)
4     // divide coverage by number of components in c2
5     m2 ← coverage/len(c2)
6 end
```

Algorithmus 3 : Heuristic to calculate m2 metric: Function `ComputeM2` in Algorithm 1

In order benefit those smells that are logically a better fit to be mapped to a certain smell variation but that are discriminated by the aforementioned aspect, the m2 metrics becomes an important factor. This

metric calculates the coverage in percent of the components entailed by a tree node compared to the current smell variation that needs to be added to the tree. The values ranges from 0 to 1 where 0 means that the two smell variations are not related to one another at all and 1 means that all components of the tree node are also present in the current smell variation. Lastly, the higher the value of m2 the more similar are the compared smell variations.

However, m2 does not consider the size of the smell which is why we combine m2 and m1 in order to determine the best parent node of the tree. As on can see in from Line 16 to 20 in Algorithm 1, the m1 and m2 metrics are calculated for every existing node of the tree. The combination of both metrics for each node are then used to determine the parent node.

This is done by first determining the best values for m1 and m2 of the given sets for those metrics. As a remainder, the lower the value for m1. For m2 it is the other way around and the highest value (not possible to be higher than 1) is the best. Subsequently, the algorithm determines if there is one or more tree nodes that have a perfect value for both m1 and m2 metrics. If there is no node that has both metrics as optimal as possible the algorithm determines the best nodes where the either m1 or m2 is has the best value. In the case there are multiple nodes with either both best values or one best value we found that using the oldest node as tree is the best way to create the tree. However, we also found that this situation , especially for both metrics the best value category, are very rarely and mostly there is a single node with a best value that can be determined as the parent node. Furthermore, the focus of this study is on the rationale for incurring a new smell. And although one can interpret the evolution smell tree as a way of “incurring" new smell (variations) to the system, our focus lays more on the root nodes as on rationale of the root nodes as on the rationale of the evolution of a smell. Finally, we found one node where the algorithm can determine the parent smell for a smell variation and hence this algorithm is capable of creating

an evolution tree for all smell variations of one AS of one smell type.

```
input 1 : Set m1 containing all values for metric1 of the
          compared tree nodes
input 2 : Set m2 containing all values for metric1 of the
          compared tree nodes
output  : parent identifier of tree node where current
          smell variation has to be aligned to
1  begin
2      // determine best value of m1
3      minM1 ← min(m1)
4      // determine best value of m2
5      maxM2 ← max(m2)
6      if NodeHasBothBest(m1, m2, minM1, maxM2) then
7          bothBestNodes ←
            GetBothBestNodes(m1, m2, minM1, maxM2)
8          parent ← GetEarliestNode(bothBestNodes)
9      else
10         oneBestNodes ←
            GetOneBestNodes(m1, m2, minM1, maxM2)
11         parent ← GetEarliestNode(oneBestNodes)
12     end
13 end
```

Algorithmus 4 : Heuristic to find parent node for smell variation: Function `FindParentNode` in Algorithm 1

In order to demonstrate the applicability of this algorithm, we provide a demonstrating example from our findings. This example uses 4 smells from smell tree 24 of the Sqoop project. Please consider these four smells with their corresponding components:

Smell 1: - `com.cloudera.sqoop.mapreduce`, `org.apache.sqoop.mapreduce`

Smell 2: - `com.cloudera.sqoop.mapreduce`, `org.apache.sqoop.mapreduce`, `com.cloudera.sqoop.mapreduce.db`, `org.apache.sqoop.mapreduce.db`

Smell 3: - `com.cloudera.sqoop.mapreduce`, `org.apache.sqoop.mapreduce`, `org.apache.sqoop.mapreduce.hcat`

Smell 4: - `com.cloudera.sqoop.mapreduce`, `org.apache.sqoop.mapreduce`, `org.apache.sqoop.mapreduce.db`, `org.apache.sqoop.mapreduce.hcat`

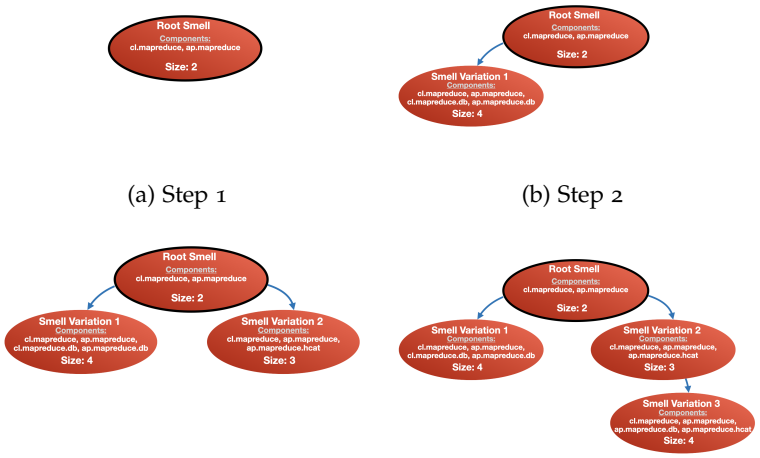
We further describe step-by-step how we align these four smells to a smell tree using our new approach. Figure 13 visualizes each of these steps:

1. Set Smell 1 as Root Smell (Figure 13a)

2. Align Smell 2 as child node to the Root Smell as Smell Variation 1 (Figure 13b)

3. Compare Smell 2 with Root Smell and Smell Variation 1:
   a) Root Smell: $m1 = 3 - 2 - 1 = 0$ and $m2 = 2/2 = 1$
   b) Smell Variation 1: $m1 = 4 - 2 - 1 = 1$ and $m2 = 1/2 = 0.5$
   c) Add Smell 2 to Root Smell as Smell Variation 2 because both metrics for the Root Smell have the best values. The metrics for Smell Variation 1 are both not optimal (Figure 13c)

4. Compare Smell 3 with Root Smell, Smell Variation 1, and Smell Variation 2:
   a) Root Smell: $m1 = 4 - 2 - 1 = 1$ and $m2 = 2/2 = 1$
   b) Smell Variation 1: $m1 = |4 - 4 - 1| = 1$ and $m2 = 3/4 = 0.75$
   c) Smell Variation 2: $m1 = 4 - 3 - 1 = 0$ and $m2 = 1/2 = 0.5$
   d) Add Smell 3 to Smell Variation 2 as Smell Variation 3. Two components have one metric best, Root Smell has m2 best and Smell Variation 2 has m1 best. Smell Variation 2 is chosen because it is the younger tree node[8]. Smell Variation 1 has no optimal value and is therefore not considered as parent node (Figure 13d)



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

Figure 13: Step 1 to 4 of creating smell tree 24 of Sqoop. There is one splitting at the root which is indicated by a black frame.

Our algorithm is based on two assumptions. First, we believe that smells can evolve. Consider a simple example of a cyclic dependency

8 Please bare in mind that the four smells have been ordered by age

with the shape of a star. If one adds a tiny cycle between a component not yet involved in this smell and the middle component of the star it will add a new beam to the it. The question is whether this new construct is a new smell or a variation of the former smell. We believe that the latter is the case which is in line with e.g. [32, 36]. A similar situation occurs in case of removing a beam from the star. In addition, similar situations can be applied to unstable dependencies and hub-like dependencies.

Our approach with aligning the evolution of smells to a tree like structure allows another interesting view on the evolution of AS. A tree suggests that a smell can evolve in two different directions which we call splitting. Consider a cycle of three components with A $\longrightarrow$ B $\longrightarrow$ C $\longrightarrow$ A. Now consider that one adds two other components between A and B which obviously results in a cycle of five components (A $\longrightarrow$ Comp1 $\longrightarrow$ Comp2 $\longrightarrow$ B $\longrightarrow$ C $\longrightarrow$ A). We found however, that sometimes the former smell (A,B,C) is not removed but remains. Further we saw that after the cycle grew to the five component cycle the original smell grew in another direction and a new components was added between A and C (A $\longrightarrow$ B $\longrightarrow$ C $\longrightarrow$ Comp3 $\longrightarrow$ A). The question that arises now is whether the three resulting smells are one smell or three smells. We understand that those smells are three different variations of the same smell because they originate from the very same smell. Figure 14 illustrates this phenomena.

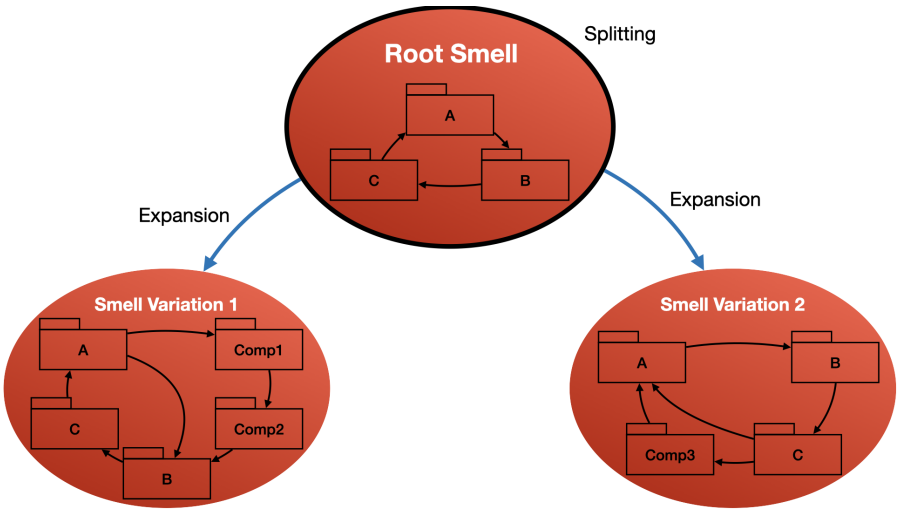


Figure 14: Three smell variation coexisting at the same time in a system

Applying this new approach to the results of ASTracker results in a list of smell trees in which one can actually observe how the smells detected are incurred in the Git diffs of the version that was found by ASTracker. Finally, the output of the pre-analysis is a list of smell trees including the root smell (the first occurrence of this smell) and

their corresponding smell variations. We will present the impact of the filtering and tree creation in Section 5.

As a conclusion of this chapter, we present an overview on the analysis progress of the pre-analysis in Table 6[9]. Here the analysed versions are depicted, in how many versions we detected smell variations, how many smell variations have been detected, and in how many smells they have been aggregated.

Projects	Analysed versions	Versions with smells	Smells Variations	Incurred Smells
Tajo	1,311	1,306	44,126	208
Tika	4,704	4,285	59,413	78
PDFBox	8,774	8,765	919,161	342
Sqoop	796	787	18,285	100
Phoenix	2,543	2,542	335,068	279
ActiveMQ	9,694	9,686	1,108,140	146

Table 6: Overview on detected smells for each project

4.4.2 *Quantitative Analysis*

The steps of the quantitative analysis are divided into collecting data from the issue and version repositories and aggregating these data into the metrics defined in Section 4.2. The data we are using here are mostly qualitative e.g. name of committer, issue type and so on. Nonetheless, we are structuring these data such that they become quantitative, i.e. accumulating the number of a specific issue type that incurs ASs. This is a valid principle for case studies as explained in [33].

4.4.2.1 *Collecting Data from Repositories*

The steps for collecting the data from the repositories are defined in Protocol 3. We first provide this protocol and describe then these tasks more in detail:

Protocol 3:

1. **Gather Github information** - request information for all smell variations using the corresponding commit sha
   a) extract the Jira issue key from the commit message
   b) extract committer and author name from GitHub

9 We only present the results of the projects that were totally analysed in this thesis. We had to discard more projects during the qualitative analysis.

2. **Check coverage of issue keys** - discard project if issue key coverage of the extracted smell variations is not at least 65 %.
3. **Gather Jira information** - store issue type and priority to csv-file
4. **Resolve commit gaps** - manually confirm the version in which the detected smell was incurred and either update information or discard smell
5. **Resolve Jira Sub-task type** - manually update issue information for all root smells with issue type *Sub-task*

Step 1: Gather Github information

The first step in collection the data required to answer the research questions defined above is to request the information from GitHub and align them to the specific smells. From the ASTracker results we received the Git commit sha for every smell variation. We use this hash value to request the commit message, the committer of this version, and the author of that commit.

In order to answer ***RQ4***, we used the committer and author name of a commit. We found that the *PyGithub API*[10] that we used to extract the information from GitHub has two ways of extracting the information for committer and author. The two developer names (committer and author) are available via `commit.committer/author` and the `commit.gitcommit.committer/author`. The difference is that the first provides the full written name (mostly first and last name) of the developer and the second is the username (mostly a nickname). However, sometimes only the gitcommit name was available. We checked and found that both names are related to the same developer. We stored both names in order to align at least one committer/author name to the version. All information derived from GitHub is stored in a csv-file.

Step 2: Check Coverage of Issue Keys

From the commit message we can extract the issue key. We do this with applying a regular expression pattern searching for the key prefix of that particular project followed by a hyphen and a number. Unfortunately, not every commit message contains the issue key. However, missing the key makes it impossible to map a version incurring ASs to a Jira issue. We checked randomly picked samples

10 https://pygithub.readthedocs.io/en/latest/introduction.html accessed on 30.07.2020

of all projects that we considered in the analysis before the analysis. Despite this we encountered the problem that for some projects the extracted smell versions only had a very low issue key coverage in the commit messages. We considered all projects as inadequate for our analysis with an issue key coverage of 65% and below and discarded these projects. The projects affected by this are: *Cocoon*, *Commons-Lang*, *Commons-Math*, *Directory Studio*, *Mina*, and *Cassandra*. We documented the coverage of issue keys in Table 7. Additionally, Table 8 shows the remaining projects that we were able to actually investigate for the AS rationale.

Project	**Coverage**	**Decisions**
Tajo	95.6%	accepted
Tika	75.3%	accepted
PDF-Box	78.4%	accepted
Sqoop	95%	accepted
Phoenix	85%	accepted
Active MQ	66%	accepted
Cocoon	0%	discarded
Commons Lang	24%	discarded
Commons Math	33%	discarded
Directory Studio	20%	discarded
Mina	17%	discarded

Table 7: Coverage of issue keys by analysed projects

Project	**Domain**	**Commits**	**Issues**	**Average LOC**	**Age**
Tajo	Data Store	2,275	2,182	257K	8.5 years
Tika	Library	4,757	3,041	160K	13.25 years
PDFBox	Library	8,998	4,775	22.2K	6 years
Sqoop	Middleware	968	3,130	328K	11 years
Phoenix	Analytics Engine	3,215	5,716	319K	6.5 years
ActiveMQ	Middleware	10,630	7,140	487K	14.5 years

Table 8: Projects analyzed in this study

Step 3: Check Coverage of Issue Keys

The final step of gathering the data for this study is to use the issue key in order to request the data from Jira. We used the *Python Jira API*[11] to request data from Jira. This way we have been able to extract the following information: *issue type*, *issue priority*, *number of comments per issue*. Similarly, to the information from GitHub we also added the

11 https://jira.readthedocs.io/en/master/ - accessed on 26.08.2020

acquired Jira information to the already existing csv-files.

Before we could actually start and analyse the data we needed to resolve two important issues: (1) resolving wrong commits due to gaps in the commit history as explained above and (2) finding the correct issue type and priority for the Jira issues labeled as *Sub-Tasks*.

Step 4: Resolve Commit Gaps

As already mentioned in our discussion about Arcan, the gaps in the commit history endanger that the data we use are biased. Indeed we found still versions mapped to a smell root where we could not find any evidence in the corresponding commit diff of that version that proofed that the smell was actually incurred with these changes. Therefore, we manually checked all commits of the root gaps which had a gap due to filtering for master branches or left out commits by Arcan.

In case we have not been able to confirm a smell in a version we searched in the parent commit to find the correct version where the smell was actually incurred. When found the correct version we manually updated the Git and Jira information. In case we could not determine the correct version where the smell was incurred we ignored the smell entirely in our analysis. However, the number of smells that we discarded turned out to be marginal compared to the total number of root smells as Table 9 indicates.

	Tajo	**Tika**	**PDFBox**	**Sqoop**	**Phoenix**	**ActiveMQ**
Gaps	97	0	28	0	34	6
Resolved	72	0	6	0	3	0
Discarded	25	0	22	0	31	6

Table 9: Information about the number root smells with gaps in the commit history, the number of resolved smells, and the number of dicarded smells

Step 5: Resolve Jira Sub-task Type

The last step in cleaning up of the dataset was to find the parent issue and thereby the correct issue type and priority for all issue with issue type *Sub-task*. For this we manually accessed the particular issue and changed the issue type and priority accordingly. We did this because *Sub-tasks* do not indicate the motivation for changing the system. Additionally, larger issues (of type *New Task*, *Improvement*, *Bug* etc.) are divided into *Sub-tasks* for means of dividing work among developers.

4.4.2.2 *Aggregating Data*

After collecting all data, one needs to aggregate them in order to answer research questions ***RQ1***, ***RQ2***, ***RQ3***, and ***RQ4*** by using the metrics defined for each question. In this section we describe how we manipulated the data in order to fit them into the metrics. This will help us subsequently to answer the research questions.

Metrics related to RQ1

RQ1 defines several metrics that we use in order to answer this question. The first metric includes the number of smells incurred in the system. We use the already described smell tree structure in order to count the number of smells. Here we simply count all root smells for every smell type. Thus, we acquire the number of smells incurred during the life-span of the project for each type. With simply summing them up we get the total number of smells in the system. Please note that we consider all smells newly incurred according to our algorithm. For this research question we do not require any information from Jira and miss-mapping of the version that incurred the smell and Jira issues does not affect the metrics required to answer ***RQ1***. Protocol 4 describes how the number of incurred smells is derived and can be found in the Appendix Section A.1.1.
The next metric concerns the number of smell variations and its characteristics. Again, we use the information from the smell trees to (1) count the number of all smell variations for each smell, (2) number of splittings and expansions, (3) the time-span during which all smell variations are added to the smell tree, and (4) characteristics about the size of the smells determined by the amount of components involved in every smell variation. The characteristics about the size of the smells are: size of the first smell variation, largest size during the evolution, smallest size during the evolution, and size of the last smell variation. Finally, we also determine whether the size of the smells is shrinking in during its evolution and whether the size actually shrinks below the size of the first small variations at some point during the evolution. Afterwards, we calculate the basic measures of central tendency and location. Protocol 5 describes the aforementioned activities (Appendix Section A.1.2).

Metrics related to RQ2 and RQ3

Since the metrics for these two research questions are similar and the aggregation of the data follows the same processes, we explain them together. The first part of these research questions is to provide an overview on the distributions of smells among smell types and issue types or issue priorities, respectively. Therefore, we simply take all the smell roots and order them by their corresponding issue type for ***RQ2*** and by issue priority for ***RQ3***. Subsequently, we count the number of

architectural smells incurred by the issues belonging to the particular issue category (e.g. bug, improvement, etc. for issue type and major, minor, etc. for issue priority) for each AS type. We found in our analysis that an issue can incur multiple smells from several types. Finally, we sum the values for all smell types such that we acquire the total amount of ASs for each issue category. Please note that we do not consider smell roots without an issue key in this analysis. Table 7 shows how many smells we consider in this part of our analysis. Protocol 6 and 7 in Appendix Section A.1.3 provide an overview on this process.

Afterwards, we process the data such that we can calculate the number of issues, sorted by issue type/priority that incur a specific smell type or combinations of it. This way we can determine whether there is a relation between a specific AS type and a specific issue type/priority. In order to achieve this we again order all smell roots by their issue type/priority and determine what type of smell each issue incurs. However, we do not count the number of smells here because finding the relation between smell type and issue type/priority, one only requires the information which smell type was incurred and not necessarily how many smell instances. We found in our analysis that an issue can incur combinations of smell types. For example, it can happen that an issue incurs one cyclic dependency and one unstable dependency. In this case we count this issue to the combination of these two smell types but not to the category cyclic dependency or unstable dependency. At last, we sum up the number of issues for each issue type/priority that incur ASs. Protocol 8 and 9 provide an overview on the described aggregation process (Appendix Section A.1.4).

Metrics related to RQ4

The metrics to answer ***RQ4*** are the number of the developers that incur a smell in a particular version and the number of smells incurred by a specific developer.

For the first one we found out that some commit versions the author is a different developer than the committer of that version. The difference is that the author in Git is the person that originally made the changes of that version the commit creates[12]. The committer on the other hand is the person who last applied this version. We conclude that if committer and author are two different developers, minimum two developers have been working on that version. Scenarios where this happens may involve changes of the original code during review by the reviewer or similar. If author and committer are the same person, than we have to assume that at least only one person

12 https://stackoverflow.com/questions/18750808/difference-between-author-and-committer-in-git accessed 05.08.2020

has been working on this code. We therefore, use the information for author and committer in order to determine how many people have been working on the code that incurs one or more ASṖrotocol 10 in Appendix Section A.1.5 provides a systematic overview on how we derived the first metric for ***RQ4***.

The second metric provides information on how many smells are incurred by a single developer. We use the information gathered to answer the first metric for ***RQ4*** and aggregate them by developer. Subsequently, we plot these information in a scatter diagram where each data point represents a developer. On the x-axis we put the number of smells a developer has and on the y-axis we put the number of commits. Although, we have information about the contributors publicly accessible, we cannot determine the developers programming experience or skills. Therefore, we use the number of commits developers have made in a particular project in order to determine their experience in that project. We assume that a developer with more commits has more knowledge and experience in a certain project than one will less commits. We use the author information in order to map a version to a specific developer since that person is the one that originally written the code. In case we could not map the developer information received via the *PyGithub API* to the developer information available on the GitHub page of the corresponding project, we discard these information. Protocol 11 provides an overview on how this metric is derived (Appendix Section A.1.6).

4.4.3 *Qualitative Analysis*

The qualitative analysis focuses on answering ***RQ5***. Here we describe how we derive the information that are defined by the metrics. Please note that the qualitative analysis is intertwined with the quantitative analysis to that extend that the activities of the quantitative analysis acquires the information such as issue types for a project that incurs architectural smells. We choose to use these information for the selection of issues that we analyse more in depth in order to extract the analysis. For example, if a project incurs smells only in *Bugs* and *Improvements* we only select those issue types for the analysis. In addition, we only analyse those issues that have the priority of the two most priority types. Moreover, we randomly select the issues from the list of smell roots following the aforementioned rules.

Metrics related to RQ5

In order to extract the metrics from the issue we first need to confirm that the version belonging to the issue actually incurs the smell. We do this by analysing the Git diff of the corresponding commit. Additionally, if available we use already compiled versions of the project

that are released after the version and visualize the dependency structure using the tool *Structure 101*[13]. This helps us to understand the relations between the packages involved in the smell. If we cannot confirm that the smell is incurred in this version we do not analyse this issue in the qualitative analysis. However, this does not necessarily mean that the smell was not incurred in that version. As the evolutionary research of architectural smells seems to be still in its infancy it is not entirely clear how to detect a new smell using the changes made in a new version of a software system. Nonetheless, confirming that the designated smell was actually incurred in this version helps not only to increase the quality of our study but also supports us with understanding the smell which is beneficial in the analysis.

We document our analysis using the tool *Atlas.ti*[14]. Here one can mark specific sentences and code them as e.g. issue context, reason for approval, or rationale for smell. This way we create transparency of our work so that others can understand our work and reproduce it. Protocol 12 provides an overview on how we conducted the quantitative analysis which are discussed subsequently in detail.

Protocol 12:

1. **Confirm smell** - check whether we can understand how the smell was incurred in this version. Discard issue if cannot be confirmed.
2. **Analyse issue description** - analyse issue description in order to understand issue context and issue/smell information
3. **Analyse discussion** - analyse the discussion on Jira in order to find evidence for incurring the AS
4. **Analyse additional sources** - include other available data sources attached to the issue
5. **Summarize findings and extract rationale** - Summarize the findings of analysing the issue and based on this and decide on the most likely rationale for incurring the smell(s)

Step 1: Confirm Smell

In order to confirmed that a smell is incurred in a specific version, we compared the affected components forming the smell with the changes made for this version commit. This is however not as straight

13 https://structure101.com/ accessed 08.08.2020
14 https://atlasti.com/ accessed 08.08.202

forward as it may appear in the first place, since every smell type manifests differently in the commit changes. The easiest one is the cyclic dependency smell. Here one merely has to check whether the components involved in the smell have been changed or smell components have been added to other smell components.

This is similar for the hub-like dependency smell because one can see if there are new dependencies added to a component or one component is added to other components. As reminder, a hub-like component is a component with many in and outgoing dependencies. Nonetheless, one risk remains for validating this smell type because in order to ensure the correctness of a smell one has to know the role of a component in a smell. An information that ASTracker does not provide. Although, this information is available in the Arcan results, it is very hard to extract them from the graph-files and is out of scope of this thesis. However, we had not many situations in which we had to manually confirm a hub-like dependency.

The unstable dependency type smell is the hardest one to confirm. This is because the smell is not always formed by adding a new dependency to a component. This can also happen when someone is removing a dependency from a component not indicated in the affected components at all. Assume that a stable component where another, more unstable component is dependent upon (is allowed by the SDP) looses some dependencies due to a change in another component. Further assume that with removing this component the stable component becomes less stable than the former unstable component (which is now more stable that the former stable component). Now a smell is formed that cannot be detected in the commit diffs of that version since the changes will only be visible in a class of the component not involved in the smell at all. We call this ripple through effect and will elaborate on this in Section and 6.

Step 2: Analyse Issue Description

After we confirmed that the smell was incurred in the smell we read through the issue description. This helps us to extract the metrics *Issue Context* and *Issue information* (type, priority, etc.). Furthermore, we use the findings from the pre-analysis to fill the *Smell Type* metric.

Step 3 & 4: Analyse Discussion & Analyse Additional Sources

We then analyse the discussion of the issue. This helps us to determine the reasons the changes of that version have been approved by the reviewers. However, the discussion behavior is different from

open source project to open source project. In some projects the discussion about the code changes are done on another platform (e.g. GitHub or ASF's review platform). If those sources are available we include them of course in our analysis.

Step 5: Summarize Findings and Extract Rationale

Finally, we try to summarize our analysis and determine the *rationale for the architectural smell*. This can be tricky because it happens that the community does not discuss the smell or even the components that are affected by the smell at all. In this case one can assume that the decision for incurring a smell had not been made deliberately. Contrary to this, not mentioning the smell is no proof on unintentionally incurring ASs. We therefore agree that no discussion on a smell merely indicates unintentionally incurring a smell but cannot hold as proof.

5 RESULTS

This section introduces all statistics and insights derived from the data collected in this study. [1]. In Section 5.1, the results that describe the evolution of ASs are presented. Section 5.2 introduces the statistics concerning the issue types that motivate developers to incur smells into software. Furthermore, Section 5.3 presents the priority that issues have that lead to ASs. In Section, 5.4 the characteristics of developers that work on a particular software version that adds smells to the system are depicted. Finally, Section 5.5 introduces the findings of the qualitative analysis concerning the rationale for incurring ASs.

5.1 SMELL EVOLUTION

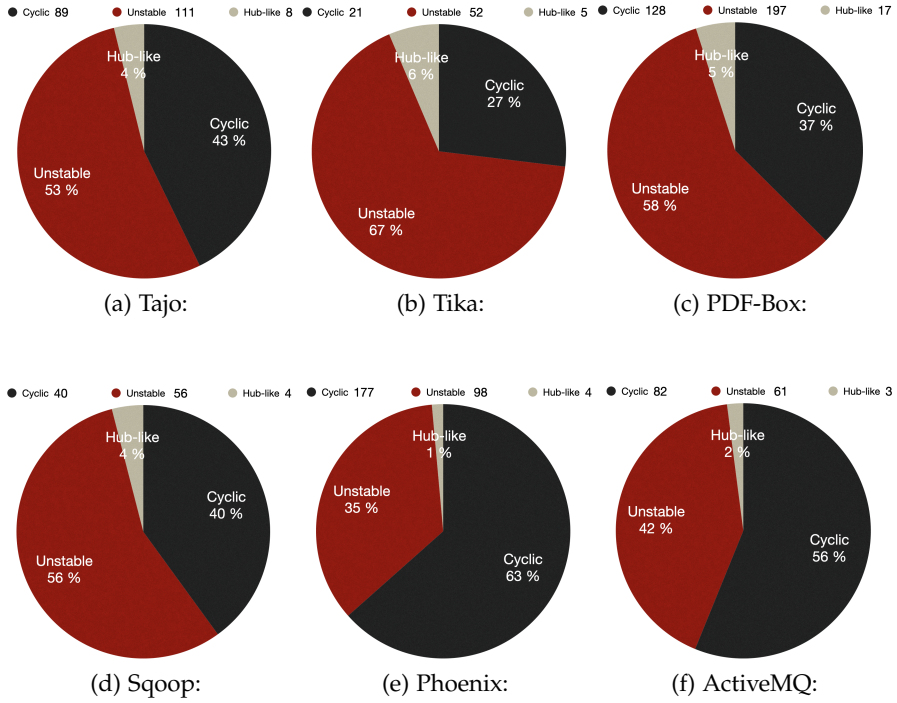
The first important step to analyse the rationales behind incurring ASs is to find the versions in which they first appear. However, these versions first need to be found. As aforementioned, the results of the tooling pipeline delivered a large amount of “new" smells for which we could not confirm that they had been incurred in that particular version as indicated by ASTracker. We also described how we created a new mapping approach in order to find all smell instances that are related to each other. We call theses instances smell variations and believe that they all belong to the very same smell instance. A good starting point to find the first version is to understand the evolution of a smell. On one hand, this allows to pinpoint the first variation of each smell (we call this variation root smell), but on the other hand, it also allows to study the evolution of each particular smell and show how it spreads throughout the system.

5.1.1 *Distribution of Smell Types*

In order to get a first overview on the architectural smells that incurred in each system, we first sorted them by their corresponding smell type. Figure 15 shows the distribution of all smells in each system. The first interesting aspect that one can see is that Hub-like Dependencies have the fewest instances in every project. They only account for one to six percent of all smell instances in the analysed projects. The majority of all smell instances is divided between Cyclic

1 All data and results that we presented here can be accessed through https://github.com/ThorstenRangnau/Jira-Project-Analyzer

and Unstable Dependencies. However, their distribution differs from project to project. Whereas Cyclic Dependencies represent between 56 and 63 percent of all smells in Phoenix and ActiveMQ, Unstable Dependencies make up between 53 to 67 percent of all smells in Tajo, Tika, PDF-Box, and Sqoop.



(a) Tajo: (b) Tika: (c) PDF-Box:

(d) Sqoop: (e) Phoenix: (f) ActiveMQ:

Figure 15: Total number of smells incurred in each project

5.1.2 *Evolution of Architectural Smells*

As previously mentioned in Section 4.4.1, our new mapping approach revealed that the evolution of smells can follow a tree-like structure. In this subsection, we present the findings that the analysis of the new structure revealed. Therefore, we first present information on the characteristics of the smell variations involved in the smell trees that have been created by our approach. These characteristics concern the number of smell variations involved in a smell tree, the number of expansions and splitting points, and the duration it takes such a tree structure to evolve. Subsequently, we present information on the size of the different smell variations related to an architectural smell instance. These findings suggest that smells not only grow but also shrink during their evolution. Afterwards, we present information on the shrinking behavior of smell variations involved in an architectural smell instance. Finally, we depict an example of a smell tree that has been detected during by our mapping approach.

5.1.2.1 *Characteristics of Smell Variations*

Several smell instances compose of more than one smell instance as can be seen in Table 10. A closer look at the mean of smell variations per smell indicates that smells seem to evolve differently from project to project. For example, Tika has the lowest average of smell variations with 2.5 while PDF-Box has the largest one with 103.4. Comparing the results of the general evolution of architectural smells with Table 8 suggests that the number of smell variations is not related to either size or age of a project. While Active MQ is the oldest and largest project analysed in this study, its average number of smell variations per smell is lower than the much younger and smaller project PDF-Box. The average number of splittings and expansions seems to be related to the mean of smell variations. The more smell variations per smell, the more splittings and expansions.

Another interesting metric is the average duration it takes for a smell to evolve. In general, the duration it takes a smell to evolve seems to be different from project to project. Also, the smells appear to evolve independent from the age of the project. Again, the smells in ActiveMQ appear to evolve in a much shorter time-span than those of PDF-Box or Phoenix, which are both younger and smaller than ActiveMQ. One may think that the evolution of the smells is connected to their size. Comparing Tika and Sqoop, as well as PDF-Box and Phoenix, shows that this does not hold. Tika has the smallest average of smell variations but the average duration of smell evolution takes more than double the time as of the smells of Sqoop, which has in general double the smell variations per smell than Tika.

Another indication that ASs expand differently from project to project can be seen in the minimum and maximum values. Here, we can see that the project with the highest mean for the number of smell variations (PDF-Box) does not have the smell with the most smell variations which is Phoenix with 8473 smell variations for a single smell. Additionally, the project with the longest duration in the evolution of smell variations (Phoenix) does not have the smell which takes the longest to evolve completely. This project is PDF-Box with 138 month.

However, one thing appears to be similar in every project and there are always smells that concise of only one smell variations. This also explains why the minimum value for splittings, expansions, and durations is always 0.

The evolution of the smell tree can happen in two different ways which we call expansion and splitting. An expansion happens when a new smell variation simply adds further components to an already existing smell variation. Splitting of the smell happens when a smell

variation adds further, but different components, to an already existing smell variation that has already expanded into another variation. The smell with the most expansions was detected in Phoenix with 6682 expansions. The smell with the most splitting can also be found in Phoenix. Here we detected a smell instance with 1685 splittings. Please note that our algorithm detects not only positive expansion, but also negative (i.e. an expansion can say that the number of components of the new smell variation may grow or shrink compared to its predecessor). Furthermore, splitting can also incur a smell variation with a smaller size than the smell variations to which it is directly related. Therefore, the high number of expansions and smells does not necessarily mean that the size of the smell variations constantly grows.

	Total			
	# smell variations	**# splitting**	**# expansions**	**duration (in month)**
Mean				
Tajo	4.7	0.8	3.1	5.3
Tika	2.5	0.2	1.4	12.9
PDF-Box	103.4	22.6	80.0	17.9
Sqoop	4.9	1.0	3.1	5.9
Phoenix	86.5	19.1	66.7	22.4
Active MQ	12.5	2.9	8.8	15.0
min/max				
Tajo	1/73	0/18	0/55	0/55
Tika	1/28	0/4	0/23	0/104
PDF-Box	1/7802	0/1685	0/6116	0/138
Sqoop	1/224	0/60	0/164	0/77
Phoenix	1/8473	0/1791	0/6682	0/68
ActiveMQ	1/542	0/134	0/408	0/122

Table 10: Evolution of smell variations

Maybe a closer look at the different smell types provides further information on the evolution of ASs. Table 11 aggregates the smell evolution by smell type. Here one can clearly see that the most smell variations are usually attached to cyclic dependencies. However, there are two exceptions. Tajo and Tika have the most smell variations allocated to the hub-like dependencies. A similar picture can be drawn

by the average number of splittings and expansions. Here, as well all projects, despite Tika, have the highest values in the cyclic dependency category.

The division into the different smell types reveal that in general the duration of the smell evolution takes the longest for that smell types that also have the highest average of smell variations. There is only one exception where this observation cannot be seen. In Phoenix, the average duration is higher for hub-like dependencies than for cyclic dependencies, although the cyclic dependencies have on average the most smell variations.

Furthermore, we can see that the smell expansions and splittings are usually higher in cyclic dependencies. The aforementioned smells with the highest expansions and splittings also belong to this smell type. Again, the only exceptions are Tajo and Tika; here, expansions and splittings are higher for the hub-like dependencies.

	Cyclic				Unstable				Hub-like			
	# smell variations	# splitting	# expansions	duration (in month)	# smell variations	# splitting	# expansions	duration (in month)	# smell variations	# splitting	# expansions	duration (in month)
Mean												
Tajo	7.7	1.6	5.4	7.9	1.8	0.2	0.8	2.8	11.5	0.8	9.9	13.5
Tika	3.1	0.4	1.8	20.6	1.3	0.1	0.4	7.6	12.6	1	10.6	35.8
PDF-Box	272.5	60.1	211.9	33.1	1.9	0.2	0.8	8.0	6.6	0.8	5.1	18.4
Sqoop	10.5	2.6	7.3	13.0	1.2	0	0.2	1.2	2	0	1	1.8
Phoenix	134.8	30.0	104.4	31.0	2.0	0.2	0.9	5.5	14.8	2.8	11.3	53.5
Active MQ	21.3	5.2	15.5	24.4	1.3	0	0.2	3.0	1.6	0	0.6	0
min/max												
Tajo	1/73	0/18	0/55	0/55	1/18	0/5	0/12	0/34	4/28	0/2	3/25	0/37
Tika	1/15	0/3	0/12	0/95	1/4	0/1	0/3	0/104	3/28	0/4	2/23	5/89
PDF-Box	1/7802	0/1685	0/6116	0/138	1/18	0/4	0/14	0/108	1/28	0/4	0/25	0/66
Sqoop	1/224	0/60	0/164	0/77	1/3	0/0	0/2	0/22	1/3	0/0	0/2	0/5
Phoenix	1/8473	0/1791	0/6682	0/68	1/45	0/10	0/35	0/65	9/22	1/5	7/17	43/68
ActiveMQ	1/542	0/134	0/408	0/122	1/5	0/1	0/3	0/49	1/3	0/0	0/2	0/0

Table 11: Evolution of smell variations by smell type

As we have already seen, the evolution of an architectural smell instance differs from project to project. This also gets evident from analysing the measures of location, created for the characteristics of the smell variations belonging to a smell tree. Those measures are depicted in Figure 16 until Figure 21. Here, the distribution of the characteristics are depicted by using boxplot with whiskers diagrams. One can find for each characteristic (variation, splitting, expansions, durations) four diagrams, one for each smell type plus one for the

accumulated smells depicted as total.

As one can see, the distribution of the values for characteristics is different for each project. This suggests that the evolution of architectural smells depends on the project itself. However, the diagrams also suggest that for almost all characteristics for each project, the distribution of the values converges towards the lower quartile or even towards the minimum value. As one can easily see, the value for the median, the lower quartile, and the minimum are often the same. In fact, that is for around 58 percent of all diagrams. Similarly, most of the rest of the diagrams have the median closer to the lower quartile than to the upper one. In addition, most of the diagrams are located at the bottom of the y-axis which suggests that the values for the measures of location mostly have a low value. Some outliers from this can be seen for hub-like dependencies in several projects such as Tajo, Tika, and Phoenix.

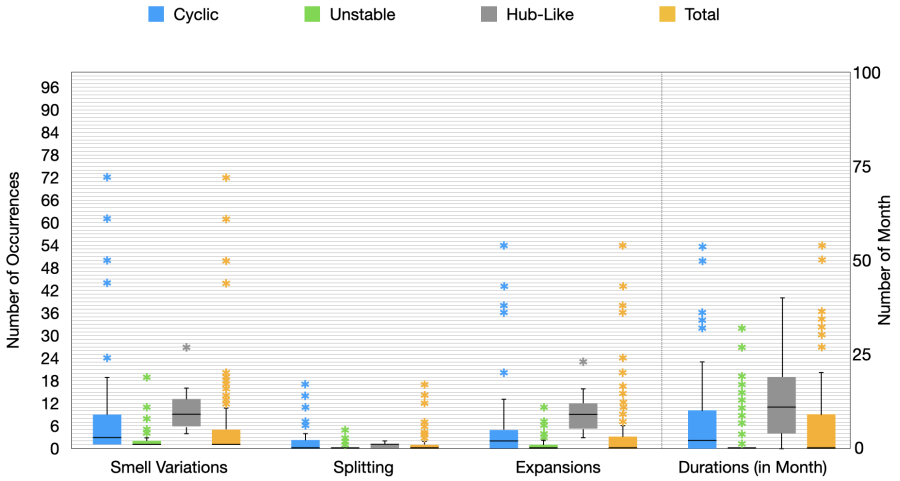


Figure 16: Distribution of the evolution of the smell tree in Tajo

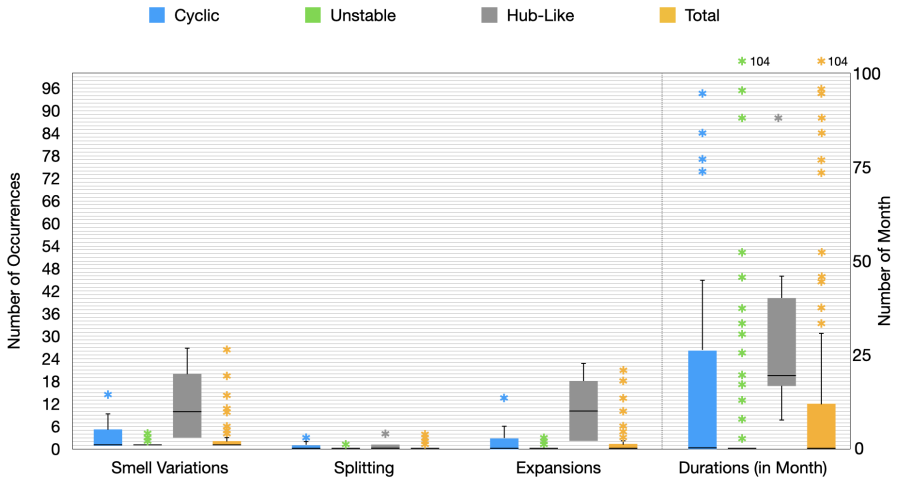


Figure 17: Distribution of the evolution of the smell tree in Tika

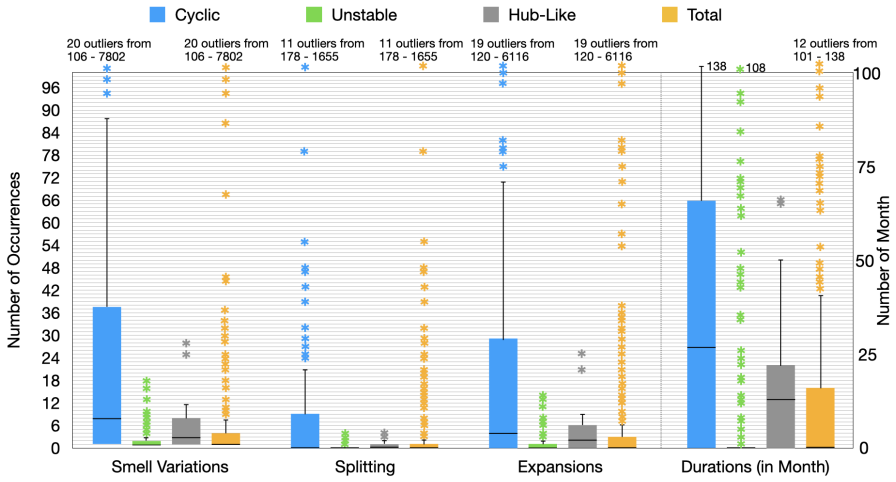


Figure 18: Distribution of the evolution of the smell tree in PDF-Box

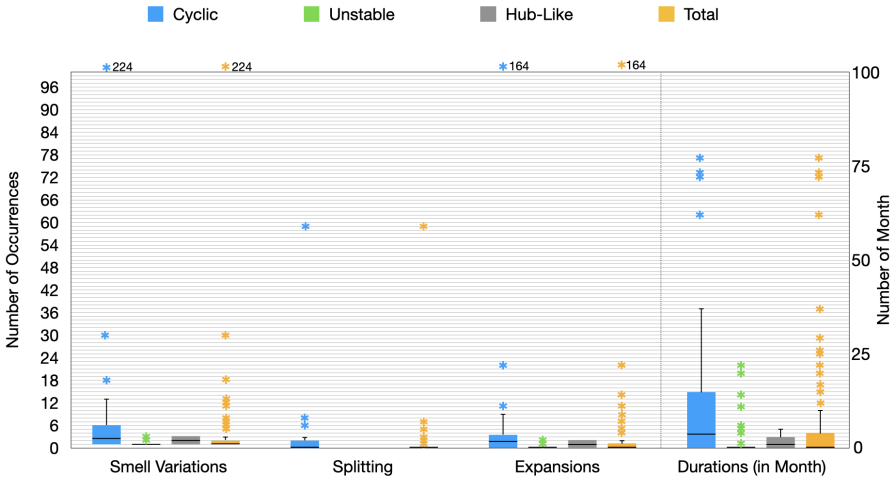


Figure 19: Distribution of the evolution of the smell tree in Sqoop

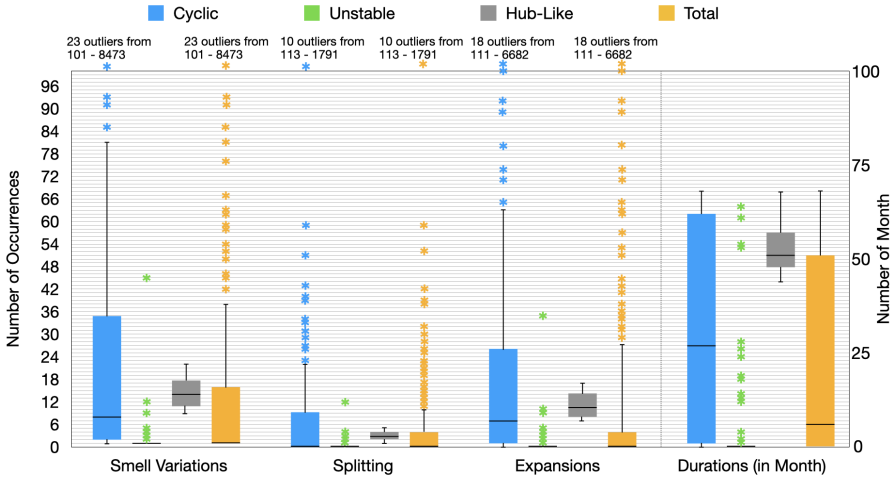


Figure 20: Distribution of the evolution of the smell tree in Phoenix

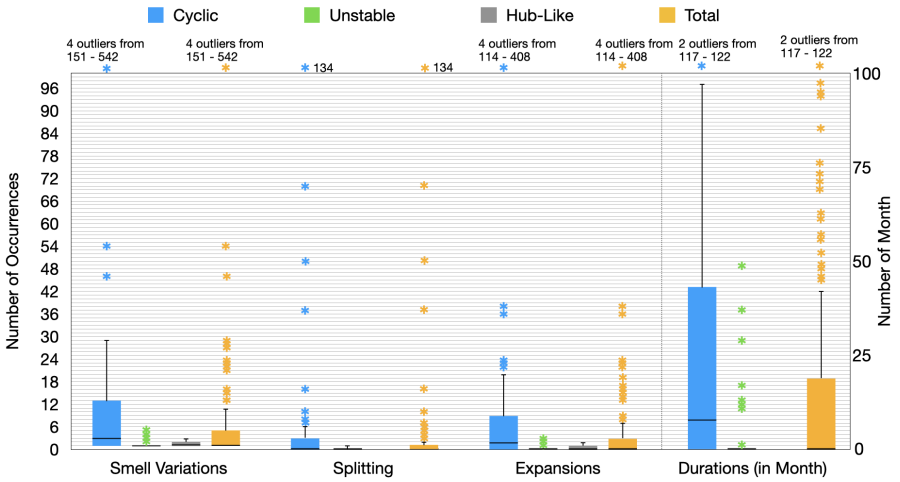


Figure 21: Distribution of the evolution of the smell tree in Active MQ

5.1.2.2 *Sizes of Smell Variation during Smell Evolution*

Also of note, the smell evolution is the size of the smell variations involved in a smell. It helps to understand how the smell evolves (i.e whether it grows or whether it shrinks over time). The size is calculated by the number of components that form the smell. Table 12 provides the mean of the component size for the first, the last, the largest, and the smallest smell variations for every project. Additionally, it also provides the minimum and maximum values of these metrics.

	Total			
	First	**Last**	**Largest**	**Smallest**
Mean				
Tajo	4.3	5.2	5.5	4.0
Tika	3.4	4.5	4.6	3.4
PDF-Box	5.1	6.2	7.1	4.3
Sqoop	3.7	4.3	4.5	3.5
Phoenix	4.3	5.8	8.0	3.6
ActiveMQ	4.1	4.9	5.6	3.7
min/max				
Tajo	2/38	2/40	2/41	2/38
Tika	2/22	2/41	2/41	2/22
PDF-Box	2/60	2/60	2/60	2/60
Sqoop	2/20	2/20	2/21	2/20
Phoenix	2/32	2/31	2/41	2/30
ActiveMQ	2/21	2/21	2/21	2/21

Table 12: Smell Variation sizes during evolution

Considering all smells together, the findings show that for all projects the average size of the first smell variation is smaller than the size of

the last one. This suggests that in general the number of components in architectural smells grow over time. However, for all projects, except Tika, the smallest smell variation is on average smaller than the first one. This is not proof and requires further investigation, but it suggests that it is possible that a smell variation shrinks in size below the size of the smell root.

The project with the highest mean for the size of the first component is PDF-Box with 5.1. The one with the smallest average is Tika with 3.4. Similarly, PDF-Box also accounts for the highest mean for the last smell variation with 6.2. The smallest mean in this category is Sqoop with 4.3. The project with the largest average smell variations is Phoenix with 8.0, and the project with the smallest average variations in this category is Sqoop with 4.5. The highest average value for the smallest smell variations can be found in PDF-Box. The smallest average in this category can be found in Tika.

Furthermore, we found that in all projects the smell with the smallest root starts with two versions. This can be expected since the architectural smell types that we investigated require a relationship between at least components. The smallest latest smell variation composes as well by two components. This does not necessarily mean that this smell actually shrinks during its evolution, but it can also be explained by a smell that starts with two components and only has one smell variation (does not evolve at all).

Comparing the four metrics for the three architectural smell types (Table 13), one can see that largest smell variations can be found in the hub-like dependencies. Cyclic dependencies appear to be the smell types with the second highest average values for the same metrics, and unstable dependencies have the lowest average values here. However, another picture can be drawn in the smell variations with the maximum values. Although, the smell with the largest first, last, largest, and smallest variation size is still a hub-like dependency, there are projects where the second largest smell in all four categories can be found in a hub-like dependency (PDF-Box).

	Cyclic				Unstable				Hub-like			
	First	Last	Largest	Smallest	First	Last	Largest	Smallest	First	Last	Largest	Smallest
mean												
Tajo	3.7	4.9	5.5	3.3	2.9	3.3	3.4	2.8	28.8	34.9	35.1	28.1
Tika	2.7	3.7	3.9	2.7	2.5	2.8	2.8	2.5	16.6	26.2	26.2	16.6
PDF-Box	4.4	6.6	7.8	3.5	3.5	3.8	4.2	2.9	28.5	31.5	35.8	26.6
Sqoop	3.7	4.9	5.3	3.8	3.0	3.2	3.3	2.9	13.25	13.75	14	13.25
Phoenix	4.6	6.7	9.9	3.6	3.0	3.3	3.5	2.8	25.2	27.8	34	22.5
ActiveMQ	4.7	6.3	7.4	4.2	2.7	2.7	2.5	2.8	14	15.3	15.3	14
min/max												
Tajo	2/9	2/11	2/12	2/9	2/9	2/9	2/9	2/8	23/28	26/40	27/41	20/38
Tika	2/4	2/7	2/8	2/4	2/6	2/7	2/7	2/6	12/22	16/41	16/41	12/22
PDF-Box	2/14	2/19	2/21	1/11	2/24	2/24	2/26	2/21	14/60	14/60	14/60	13/60
Sqoop	2/10	2/12	2/14	2/8	2/9	2/10	2/9	2/10	10/20	11/20	11/20	10/20
Phoenix	2/15	2/23	2/26	2/8	2/12	2/11	2/16	2/10	2/32	2/31	2/41	2/30
ActiveMQ	2/14	2/14	2/28	2/14	2/8	2/8	2/8	2/8	10/21	11/21	10/21	11/21

Table 13: Smell Variation sizes during evolution by smell types

5.1.2.3 *Shrinking Behavior of Smell Variations*

As mentioned above, there are indications in the data that a smell variation can shrink below the size of the smell roots. In addition, one may expect that it is more likely that smells are mostly growing during their evolution. Therefore, we want to investigate how often it occurs that a smell variation shrinks compared to its predecessor and how often a smell variation shrinks below the size of its smell root. As can be seen in Table 14, shrinking and shrinking below the size of the smell root (which we call root shrinking) happens for all smell types in all nearly all projects. The only exception for general shrinking can be found in Active MQ for hub-like dependencies. Root shrinking did not occur in Tika for cyclic dependencies and not in Active MQ for hub-like dependencies.

Project	Cyclic		Unstable		Hub-like		Total	
	Shrinking	Root Shrinking	Shrinking	Root Shrinking	Shrinking	Root Shrinking	Shrinking	Root Shrinking
Tajo	52%	26%	13%	10%	88%	25%	32%	17%
Tika	19%	0%	4%	2%	40%	20%	10%	3%
PDF-Box	56%	33%	12%	10%	53%	24%	30%	19%
Sqoop	38%	18%	4%	4%	25%	0%	18%	9%
Phoenix	70%	41%	8%	7%	75%	50%	49%	29%
ActiveMQ	54%	32%	8%	7%	0%	0%	34%	21%

Table 14: Shrinking occurrences of smell variation sizes during smell evolution

However, the amount of shrinking and root shrinking differs from project to project. In general the most shrinking can be detected in Phoenix, where 49 percent of all smells minimum shrink once during their evolution. The least shrinking can be found in Tika where only 10 percent of all smells are affected by this. Root shrinking occurs mostly in Phoenix with 29 percent and the least in Tika. Comparing the shrinking behavior of the different smell types, one can see that in these six projects the most shrinking occurs in hub-like and cyclic dependencies. In conclusion, we can see that shrinking differs form project to project. However, there are indications that may suggest that shrinking and root shrinking most likely happen for hub-like and cyclic dependencies.

5.1.2.4 *Smell Tree Examples*

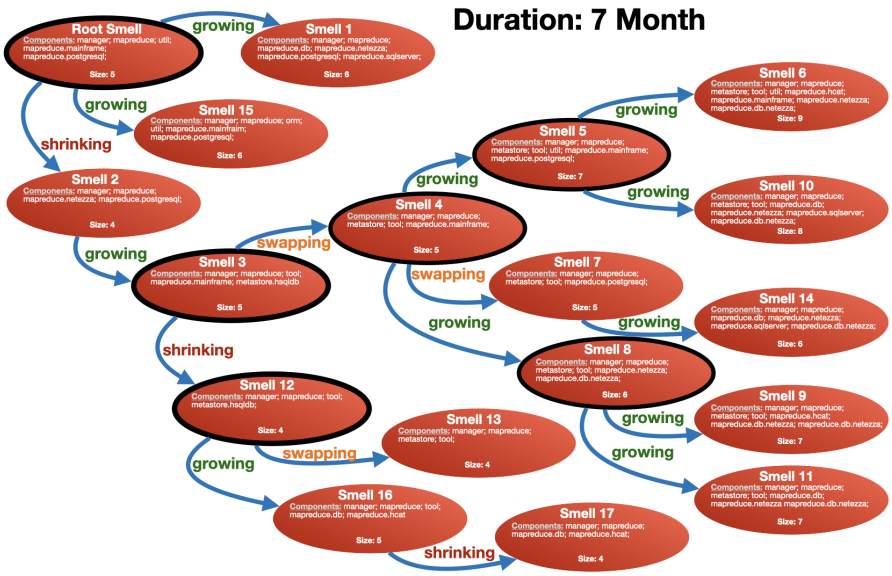


Figure 22: Smell Tree 8 for SQOOP-3273

Two examples of the smell trees that were created during this study are presented here. The first one in Figure 22 shows the tree that is created for smell 8 of the Sqoop project and is related to `SQOOP-3273`. The second one in Figure 23 for smell 76 of the Tajo project is related to `TAJO-1125`. Both figures show the different aspects of the evolution of smell variations (growing/shrinking/swapping and splitting).

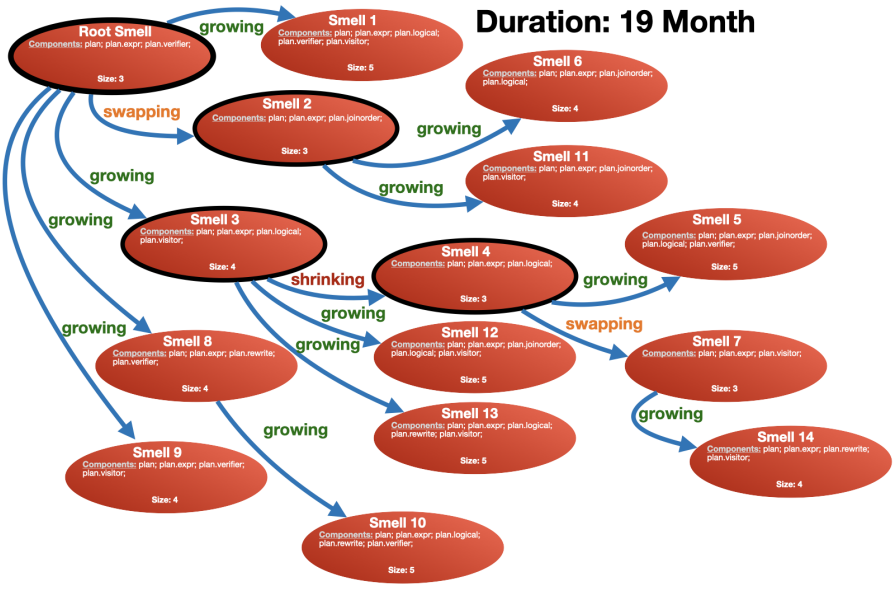


Figure 23: Smell Tree 76 for TAJO-1125

5.2 ISSUE TYPES

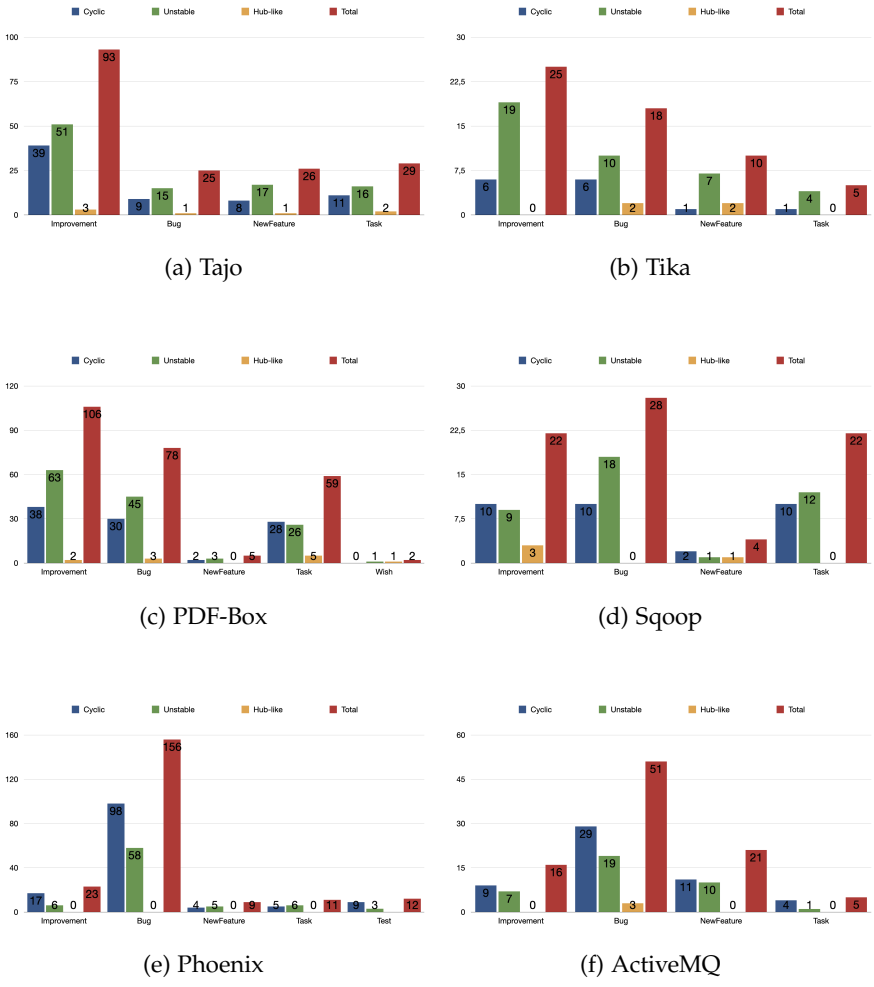
This section describes the results that are collected for the issue types. It first presents the number of smells incurred by each issue type and subsequently lays out the findings on how many issues of a certain type incur architectural smells. From our findings we understand the meaning of the issues types as follows:

1. **New Feature** - adds new functionality to the system
2. **Improvement** - improves the system in a certain quality, e.g performance
3. **Bug** - fixes an error in the system
4. **Task** - configuration management e.g. merging, release preparation
5. **Test** - demands a specific tests
6. **Wish** - wish for specific system behavior

5.2.1 *Smell Instances by Issue Type*

The first metric that concerns issue type is the number of smells that are incurred by an issue type. Figure 24 and 25 both show the results for each project. In Figure 24 we can see the absolute number of smell instances that is incurred. Furthermore, the smell instances are split by smell type. The distribution of smell instances by issue type differs from project to project. However, in three projects, the most instances

are incurred by improvements (Tajo - Figure 24a, Tika - Figure 24b, and PDF-Box - Figure 24c). In Sqoop, Phoenix, and ActiveMQ are the most smell instances incurred by the issue type “Bug" (Figure 24d, 24e, and 24f). In general, the number of smell instances of each smell type seem to follow the distribution of the total smells for each project (i.e. most total smell instances are incurred by improvements, most cyclic dependencies are incurred by improvements, etc.). Although there are minor deviations from this pattern, e.g. there is the same number of cycles incurred by improvements and bugs in Tika (Figure 24b) or more hub-like dependencies incurred by a task then by an improvements in PDF-Box (Figure 24c).



(a) Tajo

(b) Tika

(c) PDF-Box

(d) Sqoop

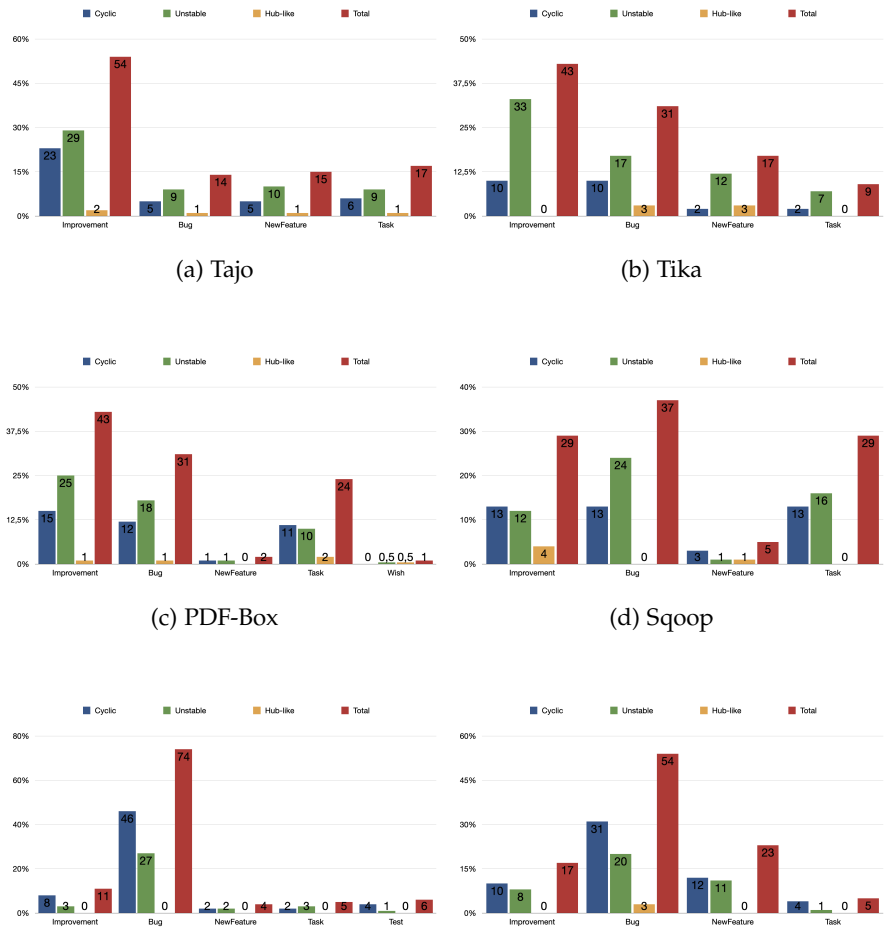
(e) Phoenix

(f) ActiveMQ

Figure 24: Total number of smells incurred by issue type

Figure 25 shows the same distribution of smell instances incurred by issue type as Figure 24 but shows the proportions of the instances in percent compared to the total number of smell instances incurred in the corresponding project. Here one can see that the ratio of total smells incurred by the issue type with the most smells is over 50% of

the total amount of smell instances for three of the projects (Tajo 25a, Phoenix 25e, and ActiveMQ 25f). For the remaining projects the ratio for the total smells of the issue type with the most smell instances converges around 40%.



(a) Tajo (b) Tika

(c) PDF-Box (d) Sqoop

(e) Phoenix (f) ActiveMQ

Figure 25: Percent of smells incurred by issue type

5.2.2 *Issue Types Incurring Architectural Smell Types*

The next metric is concerned with the number of issue types that incur a specific smell type, or combinations of smell types. This is because an issue can incur multiple smells from multiple smell types. The values are aggregated by smell type combination. These combinations are *only cyclic dependency*, *only unstable dependency*, *only hub-like dependency*, *cyclic and unstable dependency*, *cyclic and hub-like dependency*, *unstable and hub-like dependency*, and *cyclic, unstable, and hub-like dependency*. The results are presented in Table 15. It presents the total count of issues of a specific issue type. In addition to this, it presents the

normalized ratio of the issue types compared to the total count of resolved issues of this particular issue type[2].

Attr	cyclic		unstable		hub-like		cyc/un		cyc/hub		un/hub		cyc/un/hub		total	
	t	%	t	%	t	%	t	%	t	%	t	%	t	%	t	%
Tajo																
N. Feat	5	6	11	13	1	1	3	4	-	-	-	-	-	-	20	24
Impr.	15	4	18	4	-	-	69	17	-	-	-	-	2	.4	44	11
Tasks	-	-	2	1	-	-	3	2	-	-	-	-	1	.6	6	4
Bugs	7	1	7	1	1	.1	2	.3	-	-	-	-	-	-	17	3
Total	27	2	38	3	2	.1	18	1	-	-	-	-	3	.2	88	7
Tika																
N. Feat	-	-	4	2	1	.7	1	.7	-	-	1	.7	-	-	7	5
Impr.	1	.1	17	2	-	-	2	.2	-	-	-	-	-	-	20	3
Bugs	5	.5	9	1	2	.2	1	.1	-	-	-	-	-	-	17	2
Task	-	-	2	1	-	-	1	.5	-	-	-	-	-	-	3	2
Total	6	.3	32	2	3	.2	5	.2	-	-	1	.05	1	.05	48	2
PDF-Box																
Task	4	5	3	4	-	-	1	1	-	-	2	3	2	3	12	15
Wish	-	-	1	5	1	5	-	-	-	-	-	-	-	-	2	10
Impr.	13	2	20	3	2	.3	8	1	-	-	-	-	3	.4	46	7
N. Feat	1	1	2	2	-	-	1	1	-	-	-	-	-	-	4	5
Bug	18	.8	29	1	2	.09	6	.2	-	-	1	.04	-	-	56	3
Total	36	1	55	2	5	.1	16	.5	-	-	3	.09	6	.2	121	4
Sqoop																
N. Feat	1	1	1	1	1	1	-	-	-	-	-	-	-	-	3	4
Task	1	1	5	6	-	-	4	5	-	-	-	-	-	-	10	11
Impr.	4	1	1	.3	-	-	1	.3	-	-	-	-	2	.6	8	3
Bug	6	.6	8	.9	-	-	3	.3	-	-	-	-	-	-	17	2
Total	12	.8	15	1	1	.07	9	.6	-	-	-	-	2	.1	39	3
Phoenix																
N. Feat	4	4	4	4	-	-	-	-	-	-	-	-	-	-	8	9
Test	8	8	-	-	-	-	1	1	-	-	-	-	-	-	9	9
Bug	64	3	25	1	-	-	6	3	-	-	-	-	-	-	92	5
Impr.	13	3	4	.9	-	-	1	.2	-	-	-	-	-	-	18	4
Task	5	.6	2	.2	-	-	-	-	-	-	-	-	-	-	7	.9
Total	91	3	34	1	-	-	8	.2	-	-	-	-	1	.03	135	4
Active MQ																
N. Feat	4	2	5	3	-	-	3	2	-	-	-	-	-	-	12	7
Bug	22	.9	9	.4	1	.04	5	.2	-	-	2	.09	-	-	39	2
Impr.	8	.9	6	.7	-	-	1	.1	-	-	-	-	-	-	15	2
Task	2	1	1	.6	-	-	-	-	-	-	-	-	-	-	3	2
Total	36	1	21	.6	1	.03	9	.2	-	-	2	.05	-	-	69	2

Table 15: Number of issues by issue type incurring smell types or combinations of them (t). Behind each value is the normalized and rounded ratio (%) for comparing the issue types among projects. Values are normalized by using the total amount of resolved issues of that particular issue type.

2 Find the amount of issue types in the appendix in Section A.2

A first look at Table 15 already reveals a general finding for incurring architectural smells. The combination of cyclic and hub-like dependencies are never added together. In addition, the combination of unstable and hub-like dependency as well as the combination of all three smell types is incurred together only in a few projects. Only adding cyclic and unstable dependencies at the same time can be found in every project.

The number of issues that incur architectural smells differs from project to project. It varies from only two percent of all issues in Tika and ActiveMQ to seven percent in Tajo. In five projects, the issue type new feature incurs most often a smell. Only in PDF-Box are the most smells incurred through a Task. Furthermore, the distribution of the remaining issue types differs from project to project.

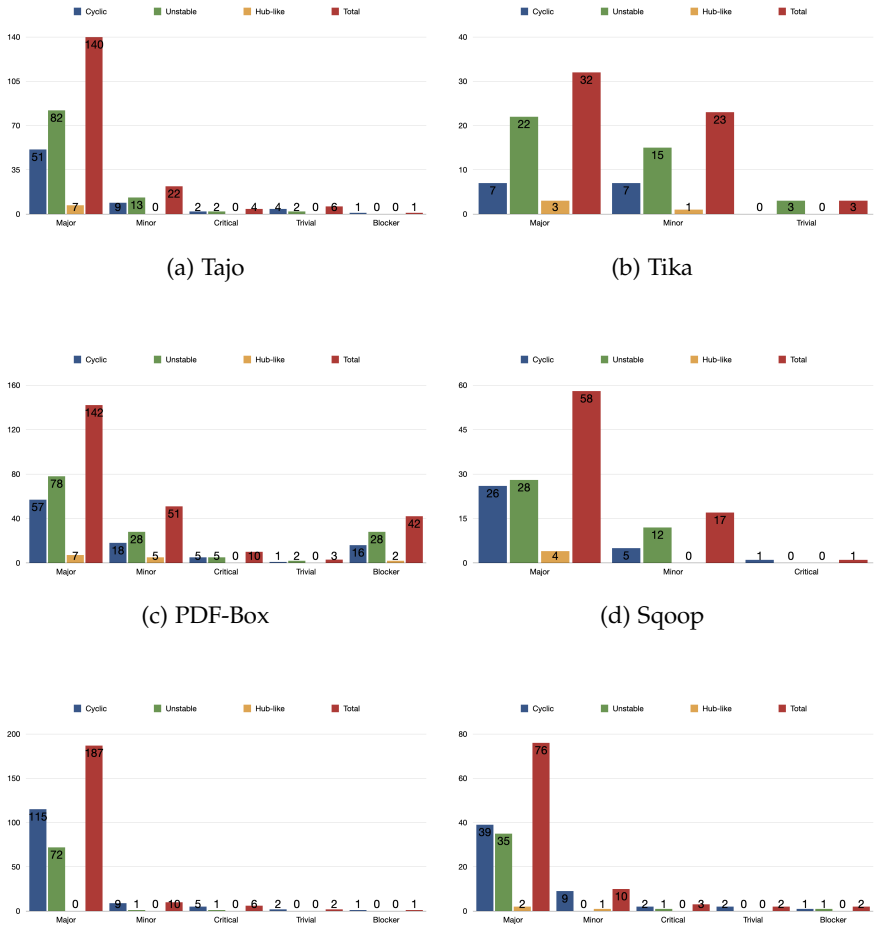
5.3 ISSUE PRIORITIES

In this section we present the findings concerning the issue priorities. First, there are the number of smell instances incurred by the issue priority. Second, we show the number of issues that incur architectural smells or combinations of it.

5.3.1 *Smell Instances by Issue Priority*

Figure 26 show the results from the number of smell instances that are incurred by issue priority. It reveals that for all projects the most smells are incurred by a major issue. The same pattern can seen for the different smell types for each project.

A closer look at the ratio of smells incurred by issue priority shows that more than fifty percent of all issues are incurred by an issue with priority level major. The lowest rate can be found in Tika with 55% and the highest rate has Phoenix with 89%. The second most number of smells is incurred by an issue with priority level minor for all project.



(a) Tajo (b) Tika

(c) PDF-Box (d) Sqoop

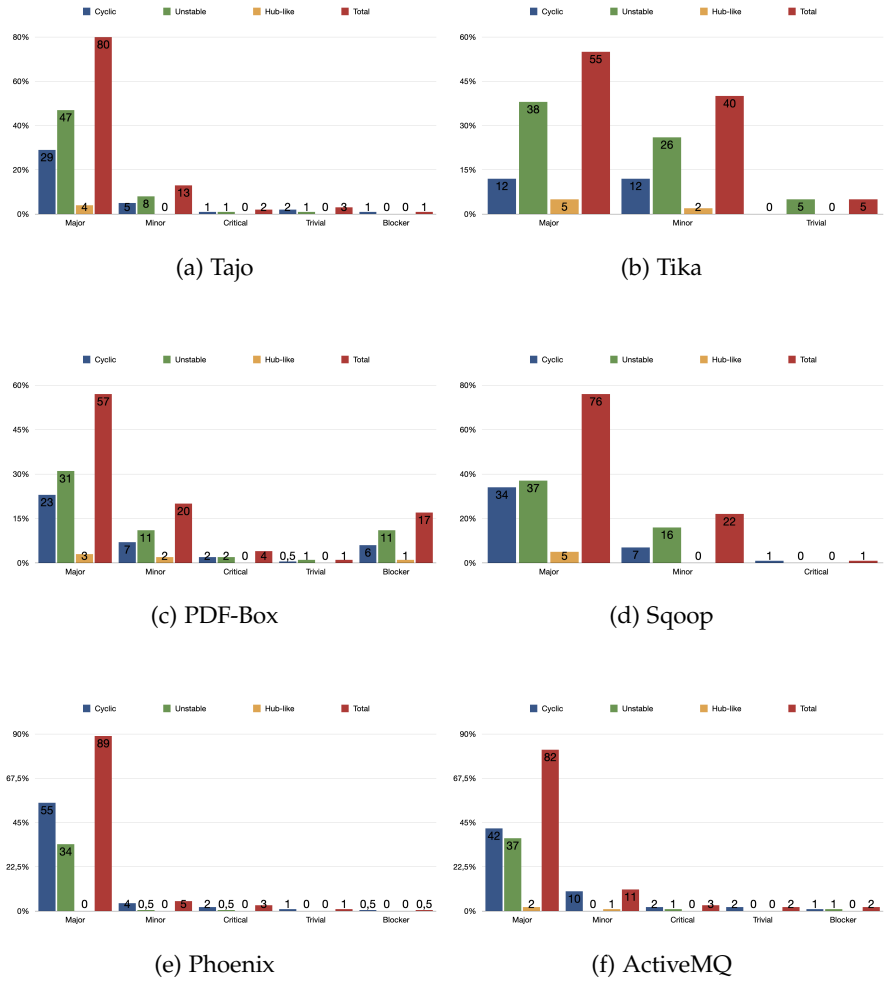
(e) Phoenix (f) ActiveMQ

Figure 26: Total number of smells incurred by issue priority

5.3.2 *Issue Priorities Incurring Architectural Smell Types*

The second metric of ***RQ3*** is concerned with the number of issue priorities that incur a specific smell type, or combinations of smell types. As already explained above, an issue can incur multiple smells of different smell types at the same time. The results are presented in Table 16. It presents the total count of issues of a specific issue priority that incurs one or more architectural smell instances. In addition to this, it presents the normalized ratio of the issue priority compared to the total count of resolved issues of this particular issue priority[3].

3 Find the amount of issue priorities in the Appendix in Section A.3



(a) Tajo

(b) Tika

(c) PDF-Box

(d) Sqoop

(e) Phoenix

(f) ActiveMQ

Figure 27: Percentage of smells incurred by issue priority

After normalizing the total issues of a corresponding issue priority that incur architectural smells, the results show that in three projects, it is more likely that smells are incurred by an issue with priority level *critical*. These projects are PDF-Box, Swoop, and Phoenix. In two projects, the priority level most likely to incur architectural smell is major. This counts for Tajo and ActiveMQ. In the remaining project Tika, the issues with priority level minor are more likely to incur new architectural smells.

Attr	cyclic		unstable		hub-like		cyc/un		cyc/hub		un/hub		cyc/un/hub		total	
	t	%	t	%	t	%	t	%	t	%	t	%	t	%	t	%
Tajo																
Major	20	2	31	4	2	.2	13	2	-	-	-	-	3	.3	69	8
Minor	3	.9	6	2	-	-	2	.5	-	-	-	-	-	-	11	3
Trivial	2	1	1	.5	-	-	1	.5	-	-	-	-	-	-	4	2
Critical	1	1	-	-	-	-	1	1	-	-	-	-	-	-	2	2
Blocker	1	2	-	-	-	-	-	-	-	-	-	-	-	-	1	2
Total	27	2	38	3	2	.1	18	1	-	-	-	-	3	.2	88	7
Tika																
Minor	3	.5	12	2	1	.2	2	.4	-	-	-	-	-	-	18	3
Major	3	.3	17	2	2	.2	3	.2	-	-	1	.09	-	-	26	2
Trivial	-	-	3	2	-	-	-	-	-	-	-	-	-	-	3	2
Total	6	.3	32	2	3	.1	5	.3	-	-	1	.05	-	-	47	2
PDF Box																
Critical	2	3	4	6	-	-	1	1	-	-	-	-	-	-	7	10
Minor	11	2	16	2	3	.4	2	.3	-	-	-	-	2	.3	34	5
Major	21	1	29	1	2	.09	10	.5	-	-	3	.1	2	.09	67	3
Trivial	1	1	2	2	-	-	-	-	-	-	-	-	-	-	3	3
Total	36	1	55	2	5	.2	15	.4	-	-	3	.09	6	.2	120	4
Sqoop																
Critical	1	4	-	-	-	-	-	-	-	-	-	-	-	-	1	4
Minor	2	.9	3	1	-	-	2	.9	-	-	-	-	-	-	7	3
Major	9	.3	12	.4	1	.03	6	.1	-	-	-	-	2	.06	30	.9
Total	12	.1	15	.2	1	.01	8	1	-	-	-	-	2	.02	39	.6
Phoenix																
Critical	5	6	1	1	-	-	-	-	-	-	-	-	-	-	6	7
Major	74	3	32	1	-	-	8	.3	-	-	-	-	-	-	114	5
Minor	8	2	1	.3	-	-	-	-	-	-	-	-	-	-	9	3
Trivial	2	3	-	-	-	-	-	-	-	-	-	-	-	-	2	3
Blocker	1	.6	-	-	-	-	-	-	-	-	-	-	-	-	1	.6
Total	91	3	34	1	-	-	8	.3	-	-	-	-	1	.03	134	4
Active MQ																
Major	24	.7	19	.6	-	-	9	.3	-	-	2	.06	-	-	54	2
Blocker	1	1	1	1	-	-	-	-	-	-	-	-	-	-	2	2
Trivial	2	1	-	-	-	-	-	-	-	-	-	-	-	-	2	1
Critical	2	.9	1	.5	-	-	-	-	-	-	-	-	-	-	3	1
Minor	7	.8	-	-	1	.1	-	-	-	-	-	-	-	-	8	.9
Total	36	.8	21	.4	1	.02	9	.2	-	-	2	.04	-	-	69	1

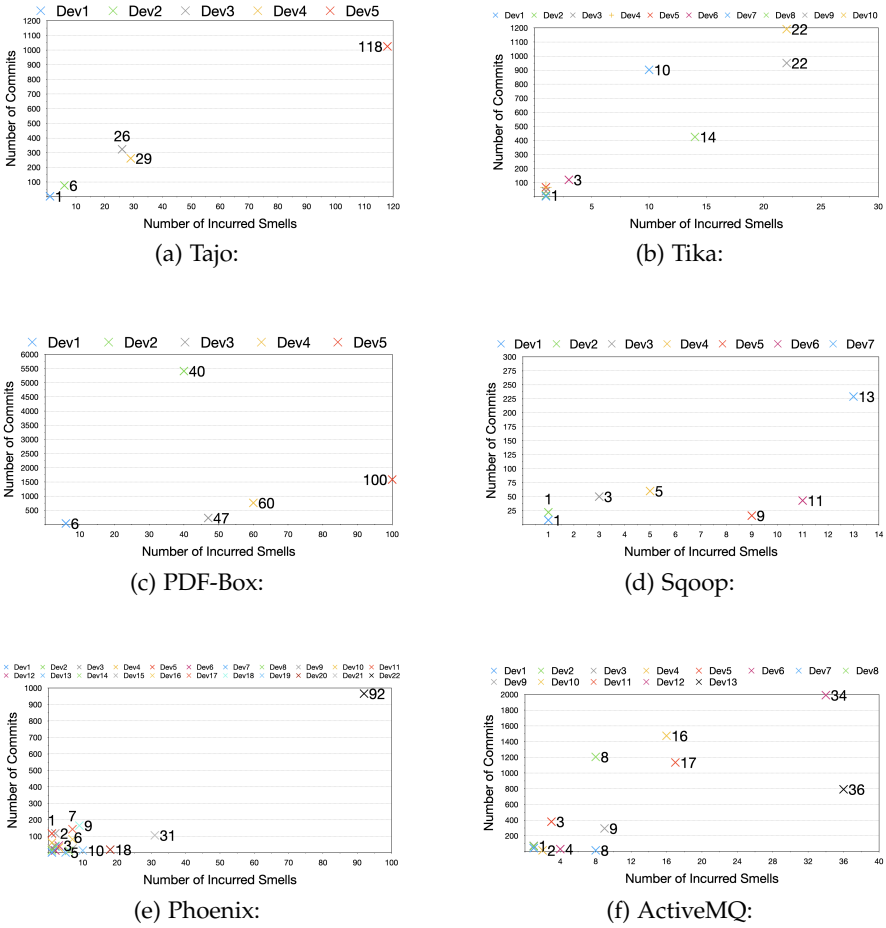
Table 16: Number of issues by priority incurring smell types or combinations of them (t). Behind each value is the normalized and rounded ratio (%) for comparing the issue priorities among projects. Values are normalized by using the total amount of resolved issues of that particular issue priority.

5.4 DEVELOPER IMPACT ON ARCHITECTURAL SMELLS

RQ4 concerns the influence that developers have on incurring architectural smells. In this section, we present the results reflected by the two metrics defined for this research question.

5.4.1 *Developer Experience Level*

This section compares the compares the experience a developer has in a project and the number of smells the same developer has incurred in this project. The results are depicted in Figure 28. There is a scatter plot for each project. Each data point is related to one developer of the corresponding project. The x-axis represents the number of incurred smells and the y-axis is the number of commits. The more to the right a data point is, the more smells this developer has incurred and the higher the data point is, the more commits this developer has added to the project.



(a) Tajo: (b) Tika:

(c) PDF-Box: (d) Sqoop:

(e) Phoenix: (f) ActiveMQ:

Figure 28: Distribution of smells incurred by developer

Comparing the results of all projects, one can see that there is a general tendency of how the data is distributed. We can see that for all projects the data follows more or less a linear dependency, i.e. the more commits a developer has, the more smells she incurs.

This effect is lucid for Tajo, Tika, and Phoenix. The data for ActiveMQ is more scattered but one can see the tendency towards a linear dependency. PDF-Box and Sqoop both have a few outliers but the tendency is still weakly linearly dependent. Without the claim for statistical significance, we calculated the correlation coefficient for every project. The results are presented in Table 17. One can identify that all values are positive. Furthermore, the tendency is also visible for the number of lines a developer has added or removed from the system. We included these attributes in order to mitigate the difference in committing behavior, i.e. one developer makes small commits while another makes huge commits. Only one project shows a weak to neutral linear dependency which is PDF-Box.

Project	**Tajo**	**Tika**	**PDF-Box**	**Sqoop**	**Phoenix**	**ActiveMQ**
commits/smells	0.99	0.92	0.1	0.4	0.92	0.74
added lines/smells	0.96	0.61	0.21	0.86	0.93	0.37
removed lines/smells	0.98	0.61	0.75	0.74	0.93	0.17

Table 17: Correlation between attributes representing the developer experience in a project

5.4.2 *Number of Developer per Smell*

Table 18 shows the distribution of the smells incurred by a single developer and by multiple developers. Here one can observe, that nearly all versions that incurred one or more smells have been created by a single developer. A few smells in Tajo and Pheonix have versions that incur architectural smells and have been worked on by multiple developers.

Project	**Single Developer**	**Multiple Developers**
Tajo	174	7
Tika	77	0
PDF-Box	319	0
Sqoop	99	0
Phoenix	231	15
Active MQ	135	5

Table 18: Overview on number of developers incurring a new architectural smell into a software project

5.5 QUALITATIVE ANALYSIS

In this section, we provide the results of the qualitative analysis. First, we provide a list of categories that include trade-off situations between a certain quality and maintenance. Subsequently, we present the findings for every issue that we analysed ordered by project.

5.5.1 *Trade-off Categories*

The qualitative analysis of 18 randomly selected issues did not reveal any discussions on incurring architectural smells. We found several issues that incur smells with the goal to add, fix, or extend functionality. In addition, we found nine issues with a certain quality objective. This is interesting because it indicates an unintentional trade-off between this particular quality and inner system quality. This is because each of these issues incurs one or more instability architectural smells. Table 19 aggregates the issues by their quality goal and provides a list of hidden trade-off categories. Some of these issues trade not only a single quality with maintainability. Therefore, several issues appear in more than one category as an example.

Category	#	Found in	Incurred Smells
performance	3	TIKA-2276, AMQ-5269, SQOOP-390	**CD:** 3, **UD:** 4, **HD:** 0
time-to-market	1	TAJO-1125	**CD:** 6, **UD:** 5, **HD:** 2
memory consumption	1	TAJO-1026	**CD:** 1, **UD:** 0, **HD:** 0
readability	1	SQOOP-3273	**CD:** 3, **UD:** 5, **HD:** 2
backwards compatibility	1	SQOOP-374	**CD:** 0, **UD:** 1, **HD:** 0
security	1	AMQ-3880	**CD:** 3, **UD:** 1, **HD:** 0
configuration management	1	TAJO-1153	**CD:** 1, **UD:** 1, **HD:** 0
code quality	1	SQOOP-3273	**CD:** 3, **UD:** 5, **HD:** 2
separation of concern	1	PDFBOX-2386	**CD:** 1, **UD:** 0, **HD:** 0
keeping legacy code	1	PHOENIX-1646	**CD:** 1, **UD:** 0, **HD:** 0

Table 19: Categories of hidden trade-offs: system quality/inner system quality

For better understanding the categories presented in the previous table, we discuss each category before we delve into the individual analysis of each of the 18 issues. We generalize these categorize from their scenario. Thus, the general implication on software maintenance become more clear.

Hidden Performance Trade-off

The hidden trade-off between performance and maintainability occurs when a change of the system has the goal of optimizes the

amount of time a system needs to perform a certain process. For example, in AMQ-5269 the developers improved sockets to close a connection immediately instead of waiting for a certain time-out runs out. This decreased the execution time of socket tests (and presumably the execution time of the affected processes at run-time). However, this trade-off incurs several smell instances the developers are not aware of.

Hidden Time-to-market Trade-off

This hidden trade-off occurs, when development is done under pressure. This happens for example when a certain change of the system is required to finish another development task. Another reason can also be that the system is changes in such a way that other tasks that are developed in parallel are hard to be integrated into these changes. This happens especially when large parts of the system's structure is changes. Developers may tend to finish these major refactorings as fast as possible in order to prevent blocking of other tasks longer than necessary. This can affect the quality of the development itself but also related processes such as code-review or testing. However, this can lead to unintentionally incur several architectural smells. We found an example for this in TAJO-1125. Here the request for a code-review was satisfied within a few hours.

Hidden Memory Consumption Trade-off

It can happen that the amount of certain data, present in the system at run-time, is limited. One way to resolve this is to store these data to e.g. a database. This may enable the system to keep its memory consumption within the prescribed boundaries. This can however lead to new architectural smell instance. In Tajo, memory limitations prevented from storing the query history. Therefore, TAJO-1026 changed the system to store this history persistently which incurred a cyclic dependency.

Hidden Readability Trade-off

It can sometimes happen that certain software constructs makes it hard for a developer to understand or read the code. Increasing the code readability can lead to incurring architectural smells. This happened in SQOOP-3273.

Hidden Backwards Compatibility Trade-off

Sometimes developers are forced to keep an old structure in order to support older, external software that relies on these old structures (backwards compatibility). However, they decide to improve the system in a certain way. Hence, they have to come up with a solution

that suits both desires (new improvement/support of legacy systems). This can lead to incur architectural smells as happened in SQOOP-374.

Hidden Security Trade-off

Security is an important issue in modern software development. Security issues in e.g. transaction protocols are often addressed by specifications. Implementing these specifications can however lead to incur smell instances (see AMQ.3880.

Hidden Configuration Management Trade-off

From time to time several (new) versions have to be aligned to be each other (merging) or the system has to be prepared for a major release. We understand this phenomenon as configuration management. Yet, the changes that are required for this can add new architectural smells.

Hidden Code Quality Trade-off

Code smells belong - as well as architectural smells - to technical debt [24]. They have a negative impact on code quality. We found a scenario where removing a huge amount of code smells have been removed during a refactoring but these changes added new architectural smell (SQOOP-3279). Here one trades code quality with architectural quality.

Hidden Separation of Concern Trade-off

It can happen that a minor system function grows during the system's evolution. In fact, this function can evolve such that it becomes so big that one wants to separate this component from the original structure where it was entailed. This follows the commonly known separation of concern principle. Though, it can happen that this separation leads to adding new smell instances. In this situation, one trades separation of concern with maintainability. We found this in PDFBOX-2386.

Hidden Keeping Legacy Code Trade-off

Sometimes a bug affects a part of the system which was formed by an mediocre or imperfect design decision. However, this legacy design decision may require to make further mediocre design decisions in order to fix the bug that affects this part of the system. One possibility may be to refactor the system to remove the first imperfect design decision and then fix the bug. However, it can be that developers decide against this because of the additional effort. They rather accept another mediocre design decision which enables them to fix

the bug in short time. It can be that this approach can lead to incur architectural smells. We found an example for this in `PHOENIX-1646`.

5.5.2 *Analysis of Individual Issues*

In this section, we present the results of the qualitative analysis for all analysed issues ordered by project. For each issue, we describe the context of the issue, the issue type and priority, the number of incurred smells, which newly added dependencies formed the smells[4], the documentation artifacts, the reason the developers approved the changes made for this issue, and the rationale for incurring the smell. In case there is no rationale mentioned in the documentation, we extracted the rationale from the information available.

5.5.2.1 *Tajo*

Issue: TAJO-1125	**Description**
Context of Issue	Over the time the project was growing, as well as the complexity of the packages logical, planner, optimizer, expressions, and expression optimizer. Since they were all part of `tajo-core` module, every client that wanted to use them was required to use the entire module. Therefore, the developers decided to separate them into its own module.
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	**CD:** 6, **UD:** 5, **HD:** 2
Smell incurred through	All components involved in the smells have been moved to the new package structure. We confirmed their existence and provide one example here for the sake of brevity: The circle between `org.apache.tajo.plan.logical`, `org.apache.tajo.plan.verifier`, `org.apache.tajo.plan.visitor` is formed since logical imports verifier, verifier imports visitor, and visitor imports logical.
Sources of Analysis	GitHub commit diffs, Jira issue
Reason for approval	The developer responsible for the changes demanded a quick review. In addition, another developer claimed that the changes being made are in need of a fast release in order to continue with another ticket. This is why they reviewed the code quickly.
Rationale for Smell	The bulk of smells is caused by the restructuring of the modules. However, the developers did not only separate the components that where described in the issue but also others. In the comments only one of these additional separations has been justified (`StorageConstants` in order to follow the localization principle. Nonetheless, there is no discussion on whether one of these new dependencies may incur any ASs. Hence, we have no evidence that the developers incurred them deliberately. In addition, the demand for the quick review may indicate that they did not have the time to be aware of the problem. We can therefore assume that the rationale for the smell is an unintentional trade-off between time-to-market and maintainability.

Table 20: TAJO-1125

4 Note: in case of too many incurred smells we only provide the proof for one or two of these smells.

Issue: TAJO-1026	Description
Context of Issue	One problem of Tajo was the way it stored the query history. This led to only keeping the very recent query history due to memory limitations. Therefore, the developers proposed to store the query on the hadoop file system.
Issue Information	**Type:** New Feature, **Priority:** Minor
Smell Type(s)	Cyclic Dependency
Smell incurred through	Adding `org.apache.tajo.worker` to `org.apache.tajo.util.history` and adding `org.apache.tajo.util.history` to `org.apache.tajo.master.querymaster`
Sources of Analysis	GitHub commit diff, GitHub pull-request review page, Jira issue
Reason for approval	The changes had been approved because the reviewer only encountered small configuration issues. He did not find the bad dependencies that formed the smell.
Rationale for Smell	The reasons why the developer incurred the bad dependencies are not discussed at all. They did not even find them in the review. We cannot find any hints for deliberately incurring the smells. Therefore, we assume that the smell was incurred unintentionally. We can see that the smell is an unintentional trade-off between memory consumption and maintenance.

Table 21: TAJO-1026

Issue: TAJO-1153	Description
Context of Issue	This issue is about configuration management for merging structure and behaviour for `off-heap tuples` into the master branch. This is because other tickets require this functionality.
Issue Information	**Type:** Task, **Priority:** Major
Smell Type(s)	1 xCyclic Dependency and 1x Unstable Dependency
Smell incurred through	**CD:** Adding `org.apache.tajo.tuple` to `org.apache.tajo.tuple.offheap` and vice versa forms the tiny cycle. Hence all dependencies that form the smell are added in this version. **UD:** The package `org.apache.tajo.tuple` is with 0.92 less stable than `org.apache.tajo.tuple.offheap` with 0.88. Yet, `offheap` depends on `tuple`
Sources of Analysis	GitHub commit diffs, Jira issue
Reason for approval	Reviewer agrees without comments. He did not find the bad dependencies
Rationale for Smell	The rationale is neither discussed nor can one find clues in the available documentation for it. We assume that the two smells are hence incurred unintentionally. This is an unintentional trade-off between configuration management and maintenance.

Table 22: TAJO-1153

5.5.2.2 *Tika*

Issue: TIKA-1010	Description
Context of Issue	In case a RTF doc embeds another document, the hex bytes of that embedded document cannot be decoded properly. This issue fixes this.
Issue Information	**Type:** Bug, **Priority:** Major
Smell Type(s)	1x Cyclic Dependency
Smell incurred through	Adding dependency to `org.apache.tika.parser.rtf` from `org.apache.tika.parser.microsoft`
Sources of Analysis	github commit, Jira Issue
Reason for approval	Fixed functionality is working and was tested. No mentioning of AS.

Table 23: TIKA-1010

Issue: TIKA-2276	Description
Context of Issue	The developers detected that the `TajoConfig`-class is created multiple times in unit testing and as a consequence the run-time is high. Therefore, they decided to reduce the run-time by reusing existing instances of `TajoConfig` where possible.
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	2x Cyclic Dependency , 1x Unstable Dependency
Smell incurred through	Adding dependency `org.apache.tika.extractor` to `org.apache.tika.parser`, already existing was `org.apache.tika.config` to `org.apache.tika.extractor` and `org.apache.tika.config` to `org.apache.tika.parser` and `org.apache.tika.parser` to `org.apache.tika.config`
Sources of Analysis	github commits, Jira Issue
Reason for approval	not discussed
Rationale for Smell	With incurring the smell the developers were able to decrease the run-time of their tests (and presumably of parsing documents in production environment) from 68 to 4 seconds. However, there is no evidence that they deliberately incur the dependency that forms the cycle or the smell itself. Therefore, one may assume that this trade-off situation was unintentional.

Table 24: TIKA-2276

Issue: TIKA-67	Description
Context of Issue	The developers have decided to add a functionality that automatically detects the document type that should be parsed and dispatches that document to the adequate parser implementation for this document type.
Issue Information	**Type:** New Feature, **Priority:** Major
Smell Type(s)	1x Unstable Dependency, 1x Hub-like Dependnency
Smell incurred through	Adding `org.apache.tika.mime` to `org.apache.tika.parser`. Most likely exceeds the median threshold for in- and outgoing dependencies in `org.apache.tika.parser` and hence forms the HD. Similarly, may `org.apache.tika.parser` be more stable than `org.apache.tika.mime`.
Sources of Analysis	GitHub commit, Jira issue
Reason for approval	Not discussed
Rationale for Smell	No rationale discussed on Jira nor indicated through the diffs on GitHub which is why we assume that the smell was incurred unintentionally

Table 25: TIKA-67

Issue: TIKA-506	Description
Context of Issue	The developers saw that there were parts of .doc and .docx files that have not been extracted from Tika at that time. Therefore, they wanted to improve the extraction of those parts.
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	1x Unstable Dependency
Smell incurred through	Adding `org.apache.tika.parser.microsoft` to `org.apache.tika.parser.microsoft.ooxml` which most likely makes the more stable stable package `ooxml` dependent on the more unstable package `microsoft`
Sources of Analysis	GitHub commit, Jira issue
Reason for approval	Not discussed, not even mentioned that it was approved
Rationale for Smell	No rationale discussed on Jira nor indicated through the diffs on GitHub which is why we assume that the smell was incurred unintentionally

Table 26: TIKA-506

5.5.2.3 *PDF Box*

Issue: PDFBOX-1689	**Description**
Context of Issue	Rendering of certain PDFs does not result in the desired output and several characters are replaced.
Issue Information	**Type:** Bug, **Priority:** Major
Smell Type(s)	1x Cyclic Dependency, 1x Unstable Dependency
Smell incurred through	**Cycle:** Add `org.apache.fontbox.ttf` to `org.apache.fontbox.util` which forms a tiny cycle. The reverse dependency already existed in `org.apache.fontbox.ttf`. **Unstable Dependency:** Most likely because `org.apache.fontbox.ttf` and `org.apache.fontbox.util.autodetect` are added to `org.apache.fontbox.util` which are both less stable than `org.apache.fontbox.util` and thereby form the smell.
Sources of Analysis	GitHub commit, Jira issue
Reason for approval	Bugfix is providing the desired functionality
Rationale for Smell	No rationale for the smell was mentioned in the sources that we found. However, the rationale for adding a new class that eventually incurred both smells is to manage font styles from the local operation system and provide an automatic way to detect them. Therefore, one can say that there was a trade-off made between maintainability and automatic customization of the system. Nonetheless, there is no evidence that the developers made this trade-off deliberately but rather unintentionally.

Table 27: PDFBOX-1689

Issue: PDFBOX-2386	**Description**
Context of Issue	As a result of the evolution of PDFBox, several components that are concerned with content streams are moved out of the util package into their own package.
Issue Information	**Type:** Improvement, **Priority:** Minor
Smell Type(s)	1x Cyclic Dependnency
Smell incurred through	Adding dependency from `org.apache.pdfbox.util` to `org.apache.pdfbox.contentstream` and vice versa.
Sources of Analysis	GitHub commit, Jira issue
Reason for approval	Not mentioned
Rationale for Smell	Not discussed at all. We therefore assume that this smell was incurred unintentionally. However, the motivation for this refactoring is to move the content stream components to the packages they belong to. We can therefore assume that it is an unintentional trade-off between separation of concern and maintainability.

Table 28: PDFBOX-2386

Issue: PDFBOX-2423	Description
Context of Issue	Major refactoring of the way PDFBox handles the page tree of the pdf. Therefore, it demands to re-write the `PDPage` component and removes the `PDPageNode` construct in order to get rid of a too low level access to raw data.
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	CD (contentstream to annotation, annotation to form, form to graphics, graphics to contentstream; various other dependencies between these packages)
Smell incurred through	Adding dependency from `org.apache.pdfbox.interactive.annotation` to `org.apache.pdfbox.contentstream` and from `org.apache.pdfbox.graphics.form` to `org.apache.pdfbox.interactive.annotation`
Sources of Analysis	GitHub commit, Jira issue
Reason for approval	Delegate to another issue because the original task was resolved but bugs remained.
Rationale for Smell	Not discussed here. We therefore assume that the smell was incurred unintentionally. The issue improves functionality.

Table 29: PDFBOX-2423

5.5.2.4 *Sqoop*

Issue: SQOOP-3273	Description
Context of Issue	A lot of Sqoop functionality is captured by classes in `com.cloudera.sqoop` packages. In order to increase readability, the developers decided to move and include all these classes to already existing Sqoop packages.
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	3x Cyclic Dependency, 5x Unstable Dependency, 2x Hub-like Dependency
Smell incurred through	**CD:** adding `org.apache.sqoop.metastore` to `org.apache.sqoop.manager`, `org.apache.sqoop.tool` to `org.apache.sqoop.metastore`, and `org.apache.sqoop` to `org.apache.sqoop.mapreduce.postgresql` forms semi clique involving `org.apache.sqoop`, `org.apache.sqoop.manager`, `org.apache.sqoop.metastore`, `org.apache.sqoop.tool`, `org.apache.sqoop.mapreduce.postgresql`, adding `org.apache.sqoop.lib` to `org.apache.sqoop.mapreduce/mapreduce.sqlserver`, `org.apache.sqoop.validation` to `org.apache.sqoop.mapreduce`, `org.apache.sqoop.mapreduce/mapreduce.db` to `org.apache.sqoop.mapreduce.sqlserver` forms semi-clique involving `org.apache.sqoop.lib`, `org.apache.sqoop.mapreduce`, `org.apache.sqoop.validation`, `org.apache.sqoop.mapreduce.db`, `org.apache.sqoop.mapreduce.sqlserver`; **UD:** aforementioned new depenedencies presumably form the five unstable dependencies since most of the components added or changed are involved in these smells [5], **HD:** similar than smells of UD[6]
Sources of Analysis	GitHub commit diff, Jira issue, apache review platform
Reason for approval	Review was satisfying after suggestions were implemented (unfortunately, no review comments available) and test passes
Rationale for Smell	The developers did not mention the problems arising from the multiple new dependencies. However, they agree that the code is now more readable and navigation has improved with the changes. Furthermore, these changes removed 2,500 code smells. Since there is no proof that the dependencies have been added deliberately, we interpret them as unintentional. Hence, this version unintentionally makes a trade-off between readability and maintenance (both inner system qualities). In addition, it swaps 2,500 code smells with 9 ASs.

Table 30: SQOOP-3273

Issue: SQOOP-390	Description
Context of Issue	Adding new functionality and increasing performance of exporting data into *PostgreSQL*
Issue Information	**Type:** New Feature, **Priority:** Major
Smell Type(s)	Unstable Dependency
Smell incurred through	Components involved in smell are `org.apache.sqoop.mapreduce`, `org.apache.sqoop.orm`, `org.apache.sqoop.util`, `org.apache.sqoop.mapreduce.db`. All packages are added to `org.apache.sqoop.mapreduce` and `org.apache.sqoop.util` which presumably means that `org.apache.sqoop.orm` and `org.apache.sqoop.mapreduce.db` are less stable than either one or both of (`mapreduce` or `util`)
Sources of Analysis	GitHub commit diff, Jira issue
Reason for approval	Not mentioned
Rationale for Smell	No discussion about why the dependencies that were added and caused the unstable dependency smell. Hence, we interpret this smell as unintentionally incurred. The smell can be seen as an unintentional trade-off between performance and maintainability.

Table 31: SQOOP-390

Issue: SQOOP-374	Description
Context of Issue	**SQOOP-369:** preparation for release of version 1.4.0, **SQOOP-390:** migrate tool packages to new space name which is a mean of preparing the release of sqoop under the Apache Incubator which requires to remove the `com.cloudera.sqoop` packages (the classes formerly capturing the functionality merely depend now on the classes of the new name-space). In addition, the classes of the old name-space are now marked as deprecated as a mean of backwards compatibility.
Issue Information	**Type:** Task (SQOOP-374 is Sub-task of SQOOP-369 which is a Task), **Priority:** Major
Smell Type(s)	Unstable Dependency
Smell incurred through	17 (deprecated) classes extends a corresponding class of `org.apache.sqoop.tool` which was maximum unstable before the changes and is depended upon `com.cloudera.sqoop.manager` and `com.cloudera.sqoop.metastore.hsqldb` which both had a certain degree of stability (0.56 `manager`, 0.75 `metastore.hsqldb`). With adding 17 hierarchy dependencies and 79 normal dependencies afferent dependencies to `org.apache.sqoop.tool` it becomes more stable than `manager` and `metastore.hsqldb` with now 0.48. However, `org.apache.sqoop.tool` also depends on `manager` and `metastore.hsqldb` which then forms the unstable dependency
Sources of Analysis	GitHub commit diff, Jira issues, namespace migration guide for sqoop[7]
Reason for approval	Code review appears to be satisfying
Rationale for Smell	The rationale for the smell is the need for backwards compatibility. The migration guide specifies the way the afferent dependencies from `org.apache.sqoop.tool` to `com.cloudera.sqoop.tool`. Although they mention the “weirdness in inheritance" (in comments of this issue), the main reason for incurring two unstable dependencies is that nobody is aware of the other dependencies that `org.apache.sqoop.tool` has to `manager` and `metastore.hsqldb`. This example may serve as an indication for ripple through effects in unstable dependencies because the changes in a package that is not involved in the smell causes it. If only the way of migrating the old packages to the new namespace would be the reason for the smell then also `org.apache.sqoop.orm` would be involved in an smell. We can therefore, consider the smell as unintentionally incurred with a trade-off between backwards compatibility and maintenance.

Table 32: SQOOP-374

5.5.2.5 *Phoenix*

Issue: PHOENIX-1646	Description
Context of Issue	Error in expression tree requires a different way of creating the indexes.
Issue Information	**Type:** Bug , **Priority:** Major
Smell Type(s)	1x Cyclic dependency
Smell incurred through	Adding dependency from `org.apache.phoenix.schema.types` to `org.apache.phoenix.parse` that closes the cycle between `org.apache.phoenix.parse`, `org.apache.phoenix.schema`, and `org.apache.phoenix.schema.types`
Sources of Analysis	GitHub Jira
Reason for approval	tests passed
Rationale for Smell	The developers do not mention the cycle that is created. However, they provide information why they add the part that forms the smell. They further mention that other ways are possible but riskier to loose information. The risk of loosing information seems to be due to older design decision (optimizations stored in constants). We can therefore assume that there is an unintentional trade-off between keeping legacy code and maintainability.

Table 33: PHOENIX-1646

Issue: PHOENIX-1514	Description
Context of Issue	Break up `PDataType` in order to prepare the system for adopting new HBase type encoding
Issue Information	**Type:** Task, **Priority:** No priority
Smell Type(s)	3x Cyclic dependency
Smell incurred through	Moving `PDataType` enum from `'org.apache.phoenix.schema` to `'org.apache.phoenix.schema.types`, which closes the cycle from those two components to `'org.apache.phoenix.exception`.
Sources of Analysis	GitHub Jira
Reason for approval	passed review and tests
Rationale for Smell	They did not discuss the smells. We therefore assume that they are unaware of the cycles. This issue improves functionality.

Table 34: PHOENIX-1514

7 https://cwiki.apache.org/confluence/display/SQOOP/Namespace+Migration - accessed 10.08.2020

5.5.2.6 *ActiveMQ*

Issue: AMQ-5591	**Description**
Context of Issue	Implementing new defined *JMS* specifications for broker side in order to allow *JMS* clients to operate with the *AMQP* (Advanced Message Queing Protocol[8])
Issue Information	**Type:** Improvement, **Priority:** Major
Smell Type(s)	1x Cyclic Dependency, 1x Unstable Dependency
Smell incurred through	**CD:** Adding `org.apache.activemq.transport.amqp.protocol` to `org.apache.activemq.transport.amqp` closes the cycle. **UD:** Presumably adding the less stable package `org.apache.activemq.transport.amqp.protocol` to the more stable package `org.apache.activemq.transport.amqp`
Sources of Analysis	GitHub commit diff, Jira issue,
Reason for approval	Not mentioned
Rationale for Smell	Unintentional - no proof for deliberate incurring smell.

Table 35: AMQ-5591

Issue: AMQ-3880	**Description**
Context of Issue	Add wss (WebSocket Secure[9]) transport
Issue Information	**Type:** New Feature, **Priority:** Major
Smell Type(s)	2x Cyclic Dependencies
Smell incurred through	Adding `org.apache.activemq.transport.https` to `org.apache.activemq.transport` which closes (1) the tiny cycle between both packages and the circle between `org.apache.activemq.transport`, `org.apache.activemq.transport.https`, and `org.apache.activemq.transport.http`
Sources of Analysis	GitHub commit diffs, Jira issue
Reason for approval	Not mentioned
Rationale for Smell	No discussion on why the dependency that created the smell is available. Hence, we have to assume that the smell was incurred unintentional. However, from the issue context we see that the smell is involved in a trade-off between maintenance and security.

Table 36: AMQ-3880

8 https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol - accessed 10.08.2020

9 https://en.wikipedia.org/wiki/WebSocket - accessed 10.08.2020

Issue: AMQ-5269	**Description**
Context of Issue	Shutting down a socket connection does not work properly which causes unnecessary delay of execution, particularly in tests.
Issue Information	**Type:** Bug, **Priority:** Major
Smell Type(s)	1x Cyclic Dependency, 2x Unstable Dependency
Smell incurred through	Adding `org.apache.activemq.transport.nio` to `org.apache.activemq.transport.tcp` which closes the circle between `org.apache.activemq.transport.tcp`, `org.apache.activemq.transport.nio`, and `org.apache.activemq.transport.broker` and presumably adds the less stable package `org.apache.activemq.transport.nio` to the more stable package `org.apache.activemq.transport.tcp`
Sources of Analysis	GitHub commit diff, Jira issue
Reason for approval	Code review appears to be satisfying and testing was successful.
Rationale for Smell	Adding the dependency that closes the circle and established the unstable dependency was not discussed. Hence, we have to assume that it was not added intentionally. Since the changes by this version allows a faster execution (especially of the tests) we interpret incurring the smell as an unintentional trade-off between performance and maintainability.

Table 37: AMQ-5269

6 DISCUSSION

This section discusses the results that have been found in this study. We therefore, answer the research questions and discuss the meaning of the findings. We present the discussion by research questions.

6.1 ARCHITECTURAL SMELL EVOLUTION

This section discusses the findings related to ***RQ1***. We present first some general findings and subsequently delve into the specific sub questions.

6.1.1 *General Findings of the Evolution of Architectural Smells*

We found that hub-like dependencies have the least occurrences in all six projects. They cover only one to six percent of all smells. This leads to the conclusion that they are not that much incurred into a software system. This results confirm with the findings of Avgeriou, Das, and Fontana in [36].

Our findings further suggest that cyclic and unstable dependencies are the most newly incurred smell types. They both share around 95 of the newly incurred smells. This tendency follows as well the findings in [36].

6.1.2 *Evolution of Architectural Smells*

We found that architectural smells evolve in a tree-like structure. More in detail, we think that this structure can only be created for cyclic dependencies. This is because on architectural level there can only be one dependency from package a to package b. This implies that a package can only have either a dependency to another package or not. Following this, two smell variations attached in a splitting to the same parent node is not possible because they would compose to the same smell variation. A “smell tree" for unstable dependencies is hence a chain without splitting. The same counts for the hub-like dependencies. Smell trees for unstable and hub-like dependencies with splitting are only possible with a focus on design level. Here one class of a package can have other in- and out-coming dependencies than another class of the same package. Only this would enable different

smell variations for unstable and hub-like dependencies.

Other interesting facts concerning the evolution of architectural smells is the shrinking and growing. We observed that the size of smell variations, measured on the number of involved components of a variation, may grow or shrink during its evolution. We further found that some smells shrink below the size of the root smell.

Unfortunately, it is out of scope of this study to investigate the reasons for this. The smell-tree evolution was not the main focus of this thesis and was only discovered by chance while investigating the birth of a smell.

6.1.3 *Size of Smell Tree Evolution*

We found that smell trees can evolve to quite a huge tree of over 8,000 smell variations. Nonetheless, we could show that these huge trees are exceptional cases, as the results show that most smell trees only compose of a few smell variation.

Another aspect that smell trees seem to have in common in the fact that cyclic dependencies appear to grow larger than unstable dependencies and hub-like dependencies. One reason for this could be the mapping that we used in this study, since we mapped the smells together using the order of the names. This works well for cycles but we have to assume that this is risky for unstable and hub-like dependencies. Without knowing which component is the stable or unstable component, our algorithm may map smells together that do no belong to each other. However, we are certain that the general idea of smell trees (or rather evolution chains) also applies to unstable and hub like dependencies. In future work we therefore recommend to map the smell variation by using the role of the components.

In contrast to the aforementioned findings, we could observe that the smell trees seem to evolve in size individually for each project. For example we could not detect any relation between the size of a project and the size of smell trees. One may assume that a tree of a larger project grows larger. However, comparing size and tree evolution of Active MQ and Phoenix reveals that this does not hold. We found similar situations while comparing other projects, e.g. Tajo and PDFBox. We can therefore conclude that smell trees evolve different from project to project.

6.1.4 *Duration of Smell Tree Evolution*

Similar to our findings concerning the evolution of the tree sizes, we can say that the duration of the smell tree evolution differs from project to project. Especially the evolution cyclic dependencies is wider spread in some projects than in others.

In addition, hub-like dependencies evolve over a very long time in Phoenix, while in Sqoop all smell trees of this smell type evolve rather quick.

One exception concerns the unstable dependencies. Here we found that the duration of the evolution converges towards zero in all projects, which means that most of the smells only composes by one smell variation and neither grow nor shrink. Yet, this behaviour could also be explained by neglecting the role of the components during the smell tree creation.

6.2 IMPACT OF ISSUE TYPES ON ARCHITECTURAL SMELLS

In this section we discuss the findings that provides an answer to ***RQ2***. This research question is subdivided into two detailed questions concerning the number of smells incurred by each issue type and the distribution of issue types that incurred new architectural smell.

6.2.1 *Number of Smells Incurred by each Issue Type*

As the results reveal, there is not a single issue type that incurred the most smells for all projects. However, one can say that the most smell instances of the analysed projects are incurred by improvements and bugs. An explanation for this is hard to determine. Yet, one can assume that there is a certain situation in the most software projects that lead developers to incur more smells than normally. Our findings suggest however, that this can hardly be explained by the issue types alone. One may argue that improvements are more likely to change the architectural structure because by its very definition it is to improve a certain quality of the system which is are primarily achieved through the architecture of a software system [5]. Yet, we found that bugs compose usually by minor changes[1]. It may even be that the smells incurred by improvements and bugs are mostly formed by building up dependencies over time and only the last “closing" dependency is added in one of these two issue types (see more in Section 6.5). Nonetheless, we need more research in this area to be able

1 Often only one class is changes and the corresponding test which should not be reflected in the dependency graph

to explain this phenomenon.

Another interesting fact is that each smell type is also incurred the most by that issue type that incurred the most total smells, i.e. the most smells are added by improvements, then the most cyclic dependencies are added by an improvement as and so on. This strengthens the assumption that there are specific situations in software developer that encourage developer to incur more smells. As well as for the total number of incurred smells above, our data does not suggest any clue why this happens.

Similarly, the fact that usually more than 40% of the smells are incurred by a single issue type (in some projects even more than 50%) also indicates the aforementioned special situations. Again, we cannot provide an explanation at this point. Furthermore, our findings may suggest this specific situation in a software project, however, as already mentioned, the issue type itself may not provide the information for this. One argument for this is the subjective developers have on the issue types. We found issues marked as improvement but were concerned with bug-fixing or a new feature that actually improved the system performance and so on.

Taking everything into consideration, we can say that there is a specific situation in every software project that leads to incurring more smells. Furthermore, this situation somehow connected to improvements and bugs. Nonetheless, the issue type itself cannot be accounted for as a general explanation for this phenomenon.

6.2.2 *Distribution of Issue Types Incurring Architectural Smells*

Our results show that the distribution on how often an issue type incurs architectural smells differs from project to project. It further shows that a new feature is that issue type that most likely incurs new ASs. This is different to the issue types that incur most smells. In order to not get confused with these two different views on the relation between issue types and smell types we present one clarifying example. Assume that a project has 100 versions where - for the sake of better understanding - every version incurs one or more smell instances. All versions are managed in 100 issues, 3 improvement, 7 new features, 90 bug fixes. In addition, there are 200 independent smell instances incurred in the 100 versions. It may further be that a single improvement incurred 100 cyclic dependencies at the same time (for the sake of brevity and better understanding we only consider cyclic dependencies here). The 4 of the 7 new features incurred together 10 cycle and each of the bug fixes adds a single smell instance to the system. In this case we can say that most of the smells

(50%) are incurred by the issue type improvement. The second most by bug fixes and the least by new features. However, it is more likely that a bug fix incurs a new smell (100%) than a new feature(57%) and an improvement(33%).

A simple explanation why new features incurs smells more likely than other issue types is that it usually adds new functionality to a system. This inevitably increases the complexity of the system because it adds more LOC to the system. In addition, this new functionality is used by other parts of the system which is realized by dependencies between the components that use this newly added functionality and the new functionality itself.

Besides this, the findings suggest that a new smell is incurred rather seldom. This is suggested by the relatively low percentage of issues that incur a certain smell[2]. This can be seen as a good sign because it implies that if one wants to prevent developers from incurring new smells she only needs to prevent these few situations. On the flip side of the coin it may be hard to determine these rare situations and their detection may be challenging.

With the results that helps answering this research question we found another interesting phenomenon. It can happen that sometimes an issue incurs a smell instance of a different smell type at the same time. Our findings show that this happens the most for the combination of cyclic and unstable dependencies but never for cyclic and hub-like dependencies. This suggest that there may be a relation between these two types that we are so far unaware of. In addition, we encounter this in all projects. The remaining combinations of smell types incurred at the same type are rather seldom and we could not find them in all projects.

6.3 IMPACT OF PRIORITIES ON ARCHITECTURAL SMELLS

In this section we discuss the findings that provides an answer to ***RQ3***. This research question is subdivided into two detailed questions concerning the number of smells incurred by each issue priority and the distribution of issue priorities that incurred new architectural smell.

6.3.1 *Number of Smells Incurred by each Issue Priority*

The results show that the most smells are incurred by an issue with priority level major in all six projects. Their ratio is minimum 55%.

2 Note: we are talking here about the smell root not smell variations which are not studied in this thesis in detail.

These findings suggest that there is something in major issues that leads developers to incurring architectural smells. However, similar to ***RQ2*** the findings do not explain why this happens. As we discussed earlier, the results regarding the issue priorities point to a specific situation in which developers incur more smell than normal. Our findings can be used to investigate this phenomenon further to narrow down situations in which developers incur more smells.

6.3.2 *Distribution of Issue Priorities Incurring Architectural Smells*

The results suggest that the priority level has a different influence among projects. Again, we have a different focus on the relation of between smell types and issue priorities as we focus here on how many issues of a certain priority incur a certain smell not how many smells are incurred by a certain issue type[3]. In some projects, major prioritized issues are more likely to incur smells and in other projects it is the priority level *critical*. An explanation for this may be subjective prioritization in each project. This suggests that the priority level is not a good attribute to determine the situation in which developers most likely incur architectural smells.

6.4 IMPACT OF DEVELOPERS ON ARCHITECTURAL SMELLS

This subsection discusses the findings of research question ***RQ4***.

6.4.1 *Experience Level of Developers in Corresponding Project*

The findings regarding the experience level of a developer in a certain project suggest that the more changes one makes the more smells she incurs. An explanation for this may be that developers are not aware of incurring architectural smells. In this case smells would be incurred as an unintentional by-product of any changes in the system. If this holds then it makes sense that the developer with the most changes in a system incurs the most smell instances.

One problem in using the number of commits as a factor for experience one has in a system is the different commit behavior of developers. Where one developer includes changes of 100 LOC in a commit another developer may include only 10 LOC in it. In this case the latter will make 10 times more commits for the same changes compared to the first developer. We mitigated this problem with including the number of added and removed LOC in calculating the coefficient.

3 Compare with the example we gave in Section 6.2.2

As it turns out all three attributes (number of commits, number of added/removed LOC) show the same tendency.

However, our findings cannot be used as proof for this because the amount of data is too little to actually calculate significant statistics on this. We therefore can use this result merely as an indicator suggesting that developers incur smells unintentionally.

6.4.2 *Number of Developers Participating in Incurring Smells*

The results suggest that versions were the code was committed by multiple developers, usually do not incur architectural smells. We found only two projects with a few versions created by multiple developers which incur architectural smells.

Nonetheless, we were not able to normalize these results. This is because requesting the information whether a commit was touched by a single or by multiple developers interferes with problems regarding rate limits of the GitHub API. Nonetheless, we manually found that there are commits with multiple developers in every project. Therefore, we can rule out that our findings are because of non-existing multiple developer commits in the projects.

Another aspect that are not reflected in the findings is that we only observed whether one or multiple developers have made changes on the code. We cannot consider the influence of discussions in code review that other developers have. For example, it is possible that only one developer works on the code and revives feed back from another developer via code review (sometimes made on platforms that are not publicly accessible).

In conclusion, we therefore see these findings as an indication that coding practices relying on team work such as pair programming may prevent incurring new smells. There needs to be more research in this area in order to make a reliable statement on this matter.

6.5 RATIONALES FOR INCURRING ARCHITECTURAL SMELLS

In this section we discuss and evaluate the findings of the qualitative analysis. We present the concept of *Hidden Trade-offs*, smells that are build up over time, and smells that are incurred at once.

6.5.1 *Hidden Trade-offs*

We did not find any proof for deliberately incurred architectural smells. Furthermore, the absence of discussions about architectural smells

suggests that developers are not aware of incurring architectural smells at all. In combination with the fact that several issues improved a certain system quality (e.g. performance, security), while they incur architectural smells at the same time, leaves us with the conclusion that developers sometimes make unintentional arrangements in which they trade their inner system quality against the quality they aim to improve. We call this phenomenon hidden trade-offs.

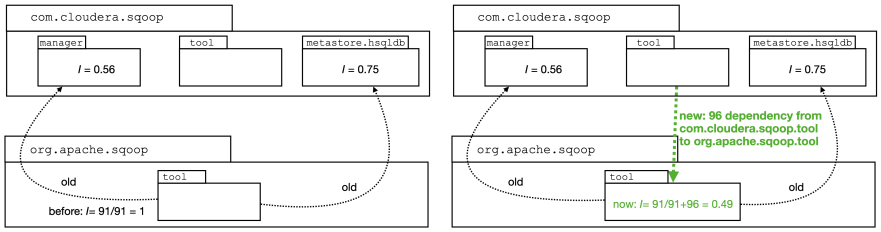
The impact of this of this observation is manifold. First, it can help to improve quality attribute driven design methods as for example described by Bass, Klein, and Bachmann in [5]. Such methods can be extended by engineering principles that assure to achieve the quality objective without decreasing inner system qualities. If this is not applicable, one can at least analyse the quality requirements for possible hidden trade-offs and consider those in technical debt management right from the beginning of the existence of the newly incurred architectural smell instances.

In addition, the findings give the research community a new impulse. It is for example possible to analyse the ratio of these hidden trade-offs. If it turns out that these trade-offs are predominantly made for a specific quality or group of qualities, one can apply this knowledge into software engineering principles and hence improve estimation techniques for architectural smells. All these means can help to prevent software practitioners to incur smells to their systems.

6.5.2 *Ripple Through Effect*

The analysis of `SQOOP-374` identified a ripple through effect for unstable dependencies. Here changes in the code in one component leads to incurring a new smell that manifests in other components. The developer may not be aware of this if she is not aware of the dependencies of the component that she imports in that part she wants to change. This implies that it is hard to identify an unstable dependency on code level. This phenomenon may serve as another indication that developers are unaware of incurring architectural smells. We depict this effect in Figure 29. Here one can see that the `org.apache.sqoop.tool` package only has dependencies to other packages but no other package depends upon it. It therefore has the stability value of 1 and is hence unstable (as explained in Section 2.2). It depends among others on the more stable packages manager and metastore.hsqldb (compare with Figure 29a). However, the changes in `SQOOP-374` require to remove several classes from com.cloudera.sqoop.tool to `org.apache.sqoop.tool`. Furthermore, in order to ensure backwards-compatibility, the old classes of `com.cloudera.sqoop.tool` extend the new classes. This way `org.apache.sqoop.tool` earns 96 outgoing de-

pendencies and turns from unstable (I = 1) to a more stable package (I = 0.49, see Figure 29b). It is now more stable than the two other packages it already depended upon which forms the unstable dependency in the end. The changes that lead to this smell are made however only in the com.cloudera.sqoop.tool package.



(a) Before (b) After

Figure 29: Ripple through effect of incurring architectural smells

6.5.3 *Building Smells Up Over Time*

As `AMQ-5269` showed, a smell is sometimes build up over a certain amount of time. As one can see in Figure 30 the circle between the three packages concise of two old dependencies and one new one. The new one inevitably closes the cycle. However, the developer responsible for this last dependency is not aware of the "half-closed" cycle while he is working on the his development task. It is therefore hard to determine for developers without further knowledge on the dependencies of the packages he is working on. This finding shows that smells build up over time and the moment where the last dependency that finishes the smell cannot be entirely hold responsible for creating the smell. In addition, the ripple through effect presented in Section 6.5.2 can also categorized as a build-up-over-time smell.

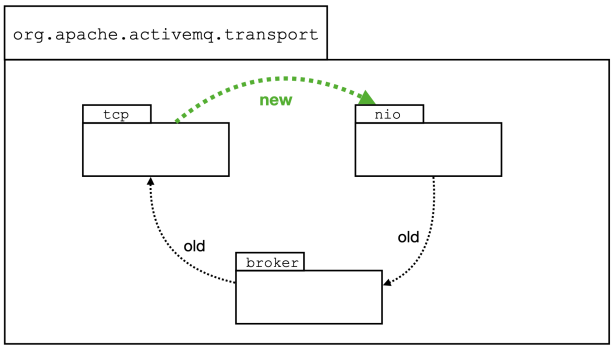


Figure 30: Building up the smell over a certain period of time

6.5.4 *Incurring a Complete Smell at Once*

A totally different situation then the two previously presented smells can be found in `TAJO-1125`. Here restructuring with separating functionality into its own package module lead to incurring a smell completely at the same time. As one can see in Figure 31, all dependencies that form the smell are added in the very same version. We believe that such a situation should to be easier to be aware of by the attentive developer. This is because she adds all dependencies at once and is hence be able to see all dependencies in her current changes. We therefore, think that being aware of an architectural smell is easier when incurred at once.

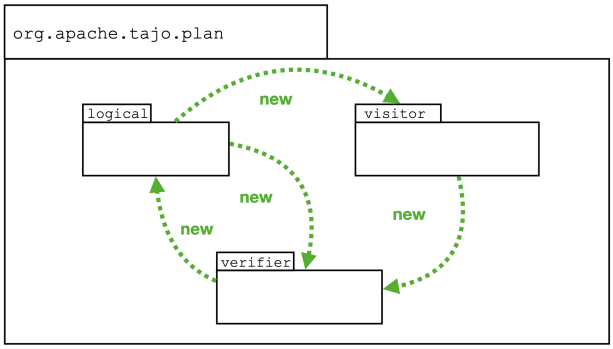


Figure 31: Incurring the entire smell at the same time

6.6 CHALLENGES

In this study we encountered several challenges and pitfalls during the analysis of the 25 original selected projects. In this section we want to discuss those challenges. This may help researchers in future work to improve studies on finding the rationale for architectural smells.

The first problem is the scalability of the available smell detection tools. Especially, analysing all versions of a software project with Arcan turned out to be time consuming. Depending on the internal structure of the project under question, we have not been able to execute the analysis on a local machine. We therefore, switched to the *Peregrine* High Performance Computation (HPC) Cluster of the University of Groningen. This cluster provides configurable execution nodes of various size. Yet, even with a huge amount of resources it was not possible to execute several projects. There are two reasons for this. First, the Arcan execution appears to be totally sequential. With increasing number of version (often accompanied by increasing project complexity) the execution takes obviously longer and longer[4]

4 Without proof we assume an exponential growth in execution with increasing project size

for every software version. Technically, one could start the execution and await the program to be terminated after a few weeks. This leads however to the second reason why it was not possible to execute all projects properly. The *HPC* limits the reservation of the running execution nodes by 240h. If this limit exceeds, the job is interrupted and the execution terminates. Fortunately, Arcan stores the graph file for every version after analysing this particular version to disc. Nonetheless, the analysis for several projects covered only a few month of the first period of the project life-span. It is therefore desirable for future work to have a smell detection tool with parallel or concurrent execution.

Another problem lies in the selection of the versions that Arcan analysis. Although configured to analyse only a certain branch (e.g. master/trunk), Arcan includes version of another branch in the analysis. One reason for this may be the software library Arcan uses for access to the Git project. Another reason may also be that most of the ASF projects are managed using *SVN* and the GitHub repository is only a mirror for the project. It is conceivable that there are problems with the transition of the project from one technology to another one (*SVN* to GitHub). However, these are only assumption and requires further investigations to fix these problems.

In addition, Arcan sometimes skips versions in its analysis. This creates gaps in the analysis of the project and leads to indicating wrong versions in which a smell is incurred. Assume that there is the version $v1$ in which a cycle is incurred. However, Arcan skips this version and continues analysing version $v2$. It will detect the cycle in $v2$ and later ASTracker marks it as incurred in $v2$. This however, biases our analysis because it inevitably maps these version to the wrong issue. In order to omit this problem we aimed to detect these gaps and manually analyse all version in order to determine the correct version in which a smell was incurred. However, this is a time consuming and tedious task. One reason that we were able to identify why Arcan is skipping some versions are exceptions during the analysis of the skipped version. One source of the exceptions are problems with the Git library checking out the next version of the project. If such an exception occurs, Arcan simply discards this version and continues with the next version.

Another problem concerning the tooling that we used is the original purpose the tooling was developed for. Especially, ASTracker was created to scan every 50's version or so of a problem. This high-level scanning allows to determine if a smell was remains or disappears and derives in information on the long-term evolution of a certain smell type. In our analysis we required ASTracker to analyse every

version though. This lead to two problems. Fist, in case the corresponding graph files created by Arcan were very big, ASTracker required a lot of resources for execution[5]. In addtion, ASTracker was created to map only already existing smells to each other if the smell is still present in the next version. If it is not it will be considered as a new smell. This results in a huge amount of architectural smell instances designated as a new smell. We find a workaround for this problem with our approach of creating the smell trees.

The next challenge that we encountered is to verify whether a project had passed the smell detection pipeline successfully. The log files from both Arcan and ASTracker are rather big. Manually, detecting problems deems to be a tedious and time consuming task. This includes for example to also verify if Arcan detected all project directories with the source code. Since, this structure differs from project to project, it is hard to verify this. Furthermore, this structure is likely to change while a project evolves. This needs to be automated if one aims to apply this case study to a larger set of projects. One way may be to automatically scan all directories that contain source code for all versions and compare them with the source code directories indicated in the log files of Arcan.

A general problem lies in the similarity mapping techniques in order to map smells from version to version. So far this happens using the names of the components involved in a smell. Both ASTracker and our approach are only using these information. In order to find a better mapping technique, we suggest to include the dependencies among packages into the mapping. Another approach may be to totally omit the need for mapping smells of different versions together, but to use the information entailed by the delta of every version, i.e. commit diffs in Git. This way one would be able to detect the smells directly when they are incurred (in the correct version). This allows to receive further information about how the smell was incurred (e.g. whether the entire cycle was added at once or "closing" a cycle via adding a new dependency to an already existing "half-cycle"). Using only the deltas for each version may also increase the performance of the analysis. The downside of this approach is however that one needs to create a complete new tooling for this. In addition, parallel or concurrent execution of such a tool may be rather ambiguous.

Lastly, the aforementioned problems of the analysis pipeline lead us to manually verify all smells in the qualitative analysis. This task is very tedious and slows this analysis remarkably down. For example,

5 It was nearly impossible to map the over 20,000 graph-files for Apache Cassandra, with a size of 8 to 14 MB each, using a high memory node of 512 GB and a run-time of 240h

detecting the ripple through effect of the unstable dependency took around twelve hours. A more reliable tooling can omit this process and allow a stronger focus on analysing the rationale for incurring a smell.

7 THREATS TO VALIDITY

In this section we present the threats to validity that may bias our findings.

7.1 SPLITTINGS FOR UNSTABLE AND HUB-LIKE DEPENDENCIES

Creating the smell trees using the heuristic presented in this study resulted in trees for unstable and hub-like dependencies that include splittings. However, on an architectural level, this is not possible. This is because packages on this level have either a dependency to another package or not. Thus, the smell tree for these two smell types has to be a chain. An exception is if someone observes these smell types on the design level. Here, there is a focus on dependencies between classes. In this case, a package can include two classes that form two different unstable or hub-like dependencies to other packages.

Nonetheless, this study is about architectural smells, and we consider the creation of smell trees (including splitting) as a threat to validity. However, this only affects the findings on the evolution of architectural instances but not the findings regarding the rationale of incurring architectural smells.

7.2 MISSING ROLE INFORMATION OF COMPONENTS

Another flaw in the creation of the smell trees is neglecting role information in aligning smell variation to a smell instance and in mapping smell variations to one another. One example may serve as an clarifying explanation. Consider the components of the two smell variations both part of an unstable dependency ['A','B','C','D'] and ['A','B','C','E']. One may think that these components belong to the same smell. However, there is no information on the roles these components have. They are only variations of the same smell if, for example, the component 'C' is the most stable component in both variations and depends on the other less stable components. If the more stable component differs for each smell variations, then they do not belong to the same smell instance.

Although this information is technically available, it is hard to extract it for every smell variation. They are entailed in the Arcan graph files for the corresponding version the smell variation was created in. It

requires a lot of effort to write a program that automatically extracts the information about the role of each component in a smell variation. We deemed it to be out of scope of this work and hence had to assume that the order of the components derived from the ASTracker output already indicated the roles.

The impact of this threat may influence finding the root smell because it can be that the wrong smell variations are aligned to the same smell instance. Unfortunately, we have no means to estimate how much smell instances are affected by this.

7.3 SIMPLE NAME SORTING

Sorting the smell variations by the names of their components name and map all variations with the same first two components together, is a simple work around. We had to find a suitable solution within the scope of our work in order to find the smell root. Although this approach subsequently resulted in our observation that smell instances evolve in a tree-like structure, it bears the risk of splitting smell variations and aligns them to different smell instances than they would logically belong to.

A more elaborated approach would have been to compare all components belonging to a smell variation and map all variations together that have the same components. However, this requires a broader knowledge about how smell variations of the same smell instance of the same smell type are related to each other which is a task for future research.

7.4 WRONG SMELL DETECTION

Another threat to the validity of our findings concerns the results created by Arcan and ASTracker. Even with mapping the smell variations to their corresponding smell instances, we sometimes found versions for smell roots that we were not able to confirm. We found, for example, commit diffs that only contained changes in `.txt`-files for documentation. However, it is impossible that those changes incur new cyclic dependency.

In the beginning of this study, when we created the pipeline to analyse the projects, we validated the findings of Arcan using Designite. This confirmed that around 91% of the smells were actually present in the system (see Appendix Section A.4). Taking this into account, we can use our results from validating the randomly selected software versions as described above to estimate an error for our findings.

Therefore, we assume that our data has an error of 9%.

It is not possible to validate all findings. This has two reasons. The first one is that it is not possible to automatically analyse all versions with Designite. Hence, there is a need to manually trigger the analysis of every software version with Designite. This is again out of the scope of this thesis. The second reason for not being able to validate all smells is that Arcan and Designite incorporate different detection mechanisms for architectural smells. Therefore, it is likely to not always find the same smells in a software version.

8

CONCLUSION

In this thesis we analysed six *Apache* open source software projects in order to determine the rationale for incurring architectural smells. With our research, we discovered a new approach to map smell instances of different software versions to the same smell. With this approach we were able to analyse over 28,000 software versions with over 62,980 smell variations related to 1,153 architectural smell instances. Furthermore, we could determine the exact version in which a new smell instance was added to the system. This enabled us to study these situations and extract information from various documentation artifact. With these information we are able to shed some light on why developers incur architectural smells into their software systems.

In our work we, discovered that architectural smells evolve in a tree-like structure involving smell variations that are related to the very same smell instance. The first smell variation - we called smell root - pinpoints the “birth" of a smell. The tree character manifest especially for cyclic dependencies and we reported smells that can spread quite far over a software system.

The qualitative analysis of the smell roots revealed that smells can be incurred in two manners. First, by adding all dependencies that form a smell in the same version. Second, to build up the dependency structure over time to finally incur the smell with adding only a small part of the smell at the end. We expect that the latter situation is more dangerous for software systems because it is harder for developers to realize that they incur architectural smell instances with their changes. We discovered one special form of building up architectural smell instances over time for unstable dependencies which we called the “ripple" through effect. We deem this effect as particularly tricky for developers because the components that form the smell are not touched in the changes of this version.

During our qualitative analysis, we could not find any proof for architectural smell instances that were incurred deliberately. However, we found that developers are most likely unaware of incurring architectural smells. Moreover, this sometimes happens in hidden trade-off situations where maintainability of the system is traded unintentionally with another system quality (e.g. performance). This suggests

that architectural smells are a by-product of software development.

Another finding suggests that developers are not aware of incurring architectural smells. Usually the developer who makes more changes in the system incurs the most smells. Oppositely, the less changes a developer makes the less smells she incurs. This can be explained with the more changes one makes the more smell instances are added. We therefore see this as a further indication that developers are unaware of architectural smells and that smells are incurred as a by-product.

We found evidence for specific situations in which architectural smells are more likely to be incurred. We also discovered that most smell instances stem from the issue types improvements and bugs with a major priority level. However, we could rule out that these issue types are more likely to incur architectural smells than other issue types.

In order to enhance the quality of research on the rationale of incurring architectural smells, we recommend further studies on more accurate smell similarity mapping techniques. We suggest to not only include the component names, but also their relationship between one another in the mapping. This will result in a better similarity mapping and can help to further the knowledge of the evolution of architectural smell instances on a lower level. With a more accurate mapping technique, one can depict the evolution of smell instances of different types on a more detailed level.

Another question that derives from our work is why architectural smells evolve in a tree-like structure. A detailed investigation on this phenomenon can help to understand why and how architectural smells spread through the system. Answering this question would provide a great benefit in preventing and managing architectural smells.

We further suggest to apply this case study to more projects. As we have determined several pitfalls and challenges in analysing the evolution of architectural smells on a detailed level, we think one is better prepared to extend the research to a larger set of open source projects. This would increase the character of the quantitative analysis and give the findings a higher empirical weight.

As our qualitative analysis needed to be cut short due to the challenges we encountered, we strongly advise to deepen this analysis. Our findings already increased our understanding of why and how architectural smells are incurred in software. Therefore, conducting this analysis on a much larger scale has great potential to reveal further secrets on the rationale for architectural smells.

Part II

APPENDIX

A

APPENDIX

A.1 PROTOCOLS

This section presents the low level protocols.

A.1.1 *Protocol 4*

1. **Count smell roots** - use the created smell trees stored in csv and count all smell roots by smell type.
2. **Aggregate information** - calculate the overall number of smells per smell type and in total.
3. **Store to csv** - store the results to csv
4. **Plot in report** - present the data in a pie chart in Section 5.

A.1.2 *Protocol 5*

1. **Count smell variations, splittings, and expansions** - determine the number of variations that belong to each smell as well as the number of splittings and expansions
2. **Calculate duration of evolution** - use the version data of the first and the last variation of each smell to determine the duration of its evolution in month.
3. **Determine smell size characteristics** - extract the start and end size of the smell variations, as well as the smallest and largest size by using the number of components involved in every smell variation.
4. **Investigate shrinking** - determine whether the size of the smell variations shrinks during its evolution by comparing the size of the previous variation with the current one. In addition, check whether the size is getting smaller than the size of the first smell variation for each smell.
5. **Calculate measures of central tendency** - calculate the mean and other statistical metrics belonging to the measures of central tendency for the information extracted in this protocol.
6. **Calculate measures of location** - calculate the min, max and other statistical metrics belonging to the measures of location for the information extracted in this protocol.

A.1.3 *Protocol 6 & 7*

Protocol 6	**Protocol 7**
1. Order all smell roots by issue type. 2. Count number of smells incurred for each smell type by each issue type. 3. Sum the number for each smell type to get total number of smells for each issue type.	1. Order all smell roots by issue priority. 2. Count number of smells incurred for each smell type by each issue priority. 3. Sum the number for each smell type to get total number of smells for each priority type.

Table 38: Protocol 6 and Protocol 7

A.1.4 *Protocol 8 & 9*

Protocol 8	**Protocol 9**
1. Order smell roots by issue type. 2. Count the number of issues of each issue type incurring an AS type or combinations of them 3. Sum the total amount of issues incurring ASs for each issue type.	1. Order smell roots by issue priority. 2. Count the number of issues of each issue priority incurring an AS type or combinations of them 3. Sum the total amount of issues incurring ASs for each issue priority.

Table 39: Protocol 8 and Protocol 9

A.1.5 *Protocol 10*

1. **Fetch developer information for each version** - Request author and committer information for every commit sha that incurs a smell
2. **Aggregate information** - in case author and committer are the same we add this to versions with smells that have been worked on by only one developer. Other wise we consider this version as to be worked on by minumum two developers
3. **Store findings** - we store the findings for each project in a csv file

A.1.6 *Protocol 11*

1. **Aggregate developer information** - aggregate the information requested for metric 1 of *RQ4* using the author information of every commit.
2. **Extract developer experience** - manually map the number of commits made by every developer in the specific project to the aggregated information
3. **Plot diagram** - we crate the diagram by using number of commits and number of smells incurred by that developer

A.2 NUMBER OF RESOLVED ISSUE TYPES BY PROJECTS

In this section we present the number of resolved issues for each project by issue type.

Project	N. Feat.	Bug	Impr	Task	Wish	Test	Total
Tajo	85	614	417	160	1	3	1,280
Tika	133	866	738	172	16	3	1,928
PDF-Box	88	2182	647	79	21	9	3,026
Sqoop	76	918	293	88	0	22	1,397
Phoenix	89	1862	409	759	3	98	3,220
ActiveMQ	176	2214	851	159	11	20	3,431

Table 40: Number of Resolved Issues for each project

A.3 RESOLVED ISSUE PRIORITIES BY PROJECTS

In this section we present the number of resolved issues for each project by issue priority.

Project	Major	Minor	Critical	Blocker	Trivial	Total
Tajo	852	345	82	41	174	1,494
Tika	1,091	557	56	57	192	1,953
PDF-Box	2,158	687	72	51	96	3,064
Sqoop	3,321	216	25	48	87	6,761
Phoenix	2,406	332	86	161	60	3,045
ActiveMQ	3,341	896	221	89	172	4,719

Table 41: Resolved Issues for each project

A.4 VALIDATION OF ARCHITECTURAL SMELLS

This subjection presents the results of introduced smells, randomly selected for different TAJO version, detected by ASTracker. These smells are then compared with an analysis of the same version by Designite. Table 42 presents the finings per version. In average, Designite confirms that more than 90 % of the smells detected by ASTracker are actual smells.

Version	CD	UD	Total	%
TAJO-843(1)	21/21	10/12	31/33	93.3%
TAJO-983(1)	16/16	1/1	17/17	100%
TAJO-591(1)	20/20	10/11	30/31	96.7%
TAJO-385(1)	14/14	8/9	22/23	95.6%
TAJO-1752(1)	27/30	30/30	57/60	95%
TAJO-2163(1)	50/53	34/35	84/88	95.4%
TAJO-1497(1)	22/27	27/27	49/54	90.7%
TAJO-1345(1)	19/27	30/32	49/59	83%
TAJO-307(1)	20/21	12/13	32/34	94.1%
TAJO-971(1)	31/33	13/14	44/47	93.6%
TAJO-1043(1)	33/33	11/11	44/44	100%
TAJO-1491(1)	25/27	16/16	41/43	95.3%
TAJO-57(1)	4/4	10/11	14/15	93.3%
TAJO-245(1)	15/17	14/14	29/31	93.5%
TAJO-129(1)	6/6	8/8	14/14	100%
TAJO-1684(1)	16/18	20/20	36/38	94.7%
TAJO-1179(1)	24/27	24/24	48/51	94.1%
TAJO-1065(1)	36/37	15/15	51/52	98%
TAJO-1939(1)	22/26	32/32	54/58	93.1%
TAJO-2168(1)	56/58	33/34	89/92	96.7%
TAJO-36(1)	3/4	10/10	13/14	92.8%
TAJO-755(1)	13/14	18/18	31/32	96.8%
TAJO-2175(1)	46/49	23/24	69/73	94.5%
TAJO-1700(1)	19/21	2/2	21/23	91.3%
TAJO-692(1)	7/8	7/7	14/15	93.3%
TAJO-371(1)	8/11	15/15	23/26	88.4%
TAJO-1581(1)	31/34	32/32	63/66	95.4%
TAJO-532(1)	11/13	18/18	29/31	93.5%

Table 42: Validation of Smells for TAJO

BIBLIOGRAPHY

[1] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin. "On the Shape of Circular Dependencies in Java Programs." In: *2014 23rd Australian Software Engineering Conference*. 2014, pp. 48–57.

[2] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)." In: *Dagstuhl Reports* 6.4 (2016). Ed. by Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman, pp. 110–138. ISSN: 2192-5283. DOI: 10.4230/DagRep.6.4.110. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6693.

[3] 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 '19. Montreal, Quebec, Canada: IEEE Press, 2019, 88–97. DOI: 10.1109/TechDebt.2019.00027. URL: https://doi.org/10.1109/TechDebt.2019.00027.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. "The Goal Question Metric Approach." In: *Encyclopedia of Software Engineering*. Wiley, 1994.

[5] Leonard J. Bass, Mark Klein, and Felix Bachmann. "Quality Attribute Design Primitives and the Attribute Driven Design Method." In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. PFE '01. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 169–186. ISBN: 3-540-43659-6. URL: http://dl.acm.org/citation.cfm?id=648114.748917.

[6] Gabriele Bavota and Barbara Russo. "A Large-Scale Empirical Study on Self-Admitted Technical Debt." In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Association for Computing Machinery, 2016, 315–326. ISBN: 9781450341868. DOI: 10.1145/2901739.2901742. URL: https://doi-org.proxy-ub.rug.nl/10.1145/2901739.2901742.

[7] Jan Bosch. "Software Architecture: The Next Step." In: *Software Architecture*. Ed. by Flavio Oquendo, Brian C. Warboys, and Ron Morrison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 194–199. ISBN: 978-3-540-24769-2.

[8] Pearl Brereton, Barbara Kitchenham, David Budgen, and Zhi Li. "Using a protocol template for case study planning." In: *Proceedings of EASE* 2008 (Jan. 2008).

[9] 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. Association for Computing Machinery, 2010, 47–52. ISBN: 9781450304276. DOI: 10.1145/1882362.1882373. URL: https://doi.org/10.1145/1882362.1882373.

[10] Dennis M. Buede. *The Engineering Design of Systems: Models and Methods*. 2nd. Wiley Publishing, 2009. ISBN: 0470164026.

[11] Ward Cunningham. "The WyCash Portfolio Management System." In: *SIGPLAN OOPS Mess.* 4.2 (Dec. 1992), 29–30. ISSN: 1055-6400. DOI: 10.1145/157710.157715. URL: https://doi.org/10.1145/157710.157715.

[12] Davide Falessi, Lionel Briand, Giovanni Cantone, Rafael Capilla, and Philippe Kruchten. "The Value of Design Rationale Information." In: *ACM Transactions on Software Engineering and Methodology* To appear (http://tosem.acm.org/index.php) (Dec. 2013). DOI: 10.1145/2491509.2491515.

[13] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. "Arcan: A Tool for Architectural Smells Detection." In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 282–285.

[14] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni. "Automatic Detection of Instability Architectural Smells." In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 433–437.

[15] Martin Fowler. *TechnicalDebtQuadrant*. 2009. URL: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html (visited on 06/06/2020).

[16] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. "Identifying Architectural Bad Smells." In: *2009 13th European Conference on Software Maintenance and Reengineering*. 2009, pp. 255–258.

[17] 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: Architectures for Adaptive Software Systems*. QoSA '09. Springer-Verlag, 2009, 146–162. ISBN: 9783642023507. DOI: 10.1007/978-3-642-02351-4_10. URL: https://doi.org/10.1007/978-3-642-02351-4_10.

[18] I. Groher and R. Weinreich. "A Study on Architectural Decision-Making in Context." In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*. 2015, pp. 11–20.

[19] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[20] "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary." In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[21] Anton Jansen and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions." In: *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. WICSA '05. USA: IEEE Computer Society, 2005, 109–120. ISBN: 0769525482. DOI: 10.1109/WICSA.2005.61. URL: https://doi.org/10.1109/WICSA.2005.61.

[22] Philippe Kruchten. "An Ontology of Architectural Design Decisions in Software-Intensive Systems." In: *2nd Groningen Workshop on Software Variability* (Jan. 2004).

[23] Philippe Kruchten, Patricia Lago, and Hans van Vliet. "Building up and Reasoning about Architectural Knowledge." In: *Proceedings of the Second International Conference on Quality of Software Architectures*. QoSA'06. Springer-Verlag, 2006, 43–58. ISBN: 3540488197. DOI: 10.1007/11921998_8. URL: https://doi.org/10.1007/11921998_8.

[24] Zengyang Li, Paris Avgeriou, and Peng Liang. "A Systematic Mapping Study on Technical Debt and Its Management." In: *Journal of Systems and Software* (Dec. 2014). DOI: 10.1016/j.jss.2014.12.027.

[25] Robert C. Martin. "Object oriented design quality metrics: an analysis of dependencies." In: 1994. URL: https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf.

[26] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[27] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. "Mapping architectural decay instances to dependency models." In: *2013 4th International Workshop on Managing Technical Debt (MTD)*. 2013, pp. 39–46.

[28] V. Okanović. "Designing a web application framework." In: *2011 18th International Conference on Systems, Signals and Image Processing*. June 2011, pp. 1–4.

[29] I. Ozkaya, R. L. Nord, and P. Kruchten. "Technical Debt: From Metaphor to Theory and Practice." In: *IEEE Software* 29.06 (Nov. 2012), pp. 18–21. ISSN: 0740-7459. DOI: 10.1109/MS.2012.167.

[30] David Lorge Parnas. "Software Aging." In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. IEEE Computer Society Press, 1994, 279–287. ISBN: 081865855X.

[31] A. Potdar and E. Shihab. "An Exploratory Study on Self-Admitted Technical Debt." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 91–100.

[32] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni. "Towards an Architectural Debt Index." In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2018, pp. 408–416.

[33] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering – Guidelines and Examples*. Feb. 2012. DOI: 10.1002/9781118181034.

[34] E. d. S. Maldonado and E. Shihab. "Detecting and quantifying different types of self-admitted technical Debt." In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 2015, pp. 9–15.

[35] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. "Refactoring for Software Architecture Smells." In: *Proceedings of the 1st International Workshop on Software Refactoring*. IWoR 2016. Association for Computing Machinery, 2016, 1–4. ISBN: 9781450345095. DOI: 10.1145/2975945.2975946. URL: https://doi-org.proxy-ub.rug.nl/10.1145/2975945.2975946.

[36] D. Sas, P. Avgeriou, and F. Arcelli Fontana. "Investigating Instability Architectural Smells Evolution: An Exploratory Case Study." In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 557–567.

[37] Carolyn Seaman, Robert L. Nord, Philippe Kruchten, and Ipek Ozkaya. "Technical Debt: Beyond Definition to Understanding Report on the Sixth International Workshop on Managing Technical Debt." In: *SIGSOFT Softw. Eng. Notes* 40.2 (Apr. 2015), pp. 32–34. ISSN: 0163-5948. DOI: 10.1145/2735399.2735419. URL: http://doi.acm.org/10.1145/2735399.2735419.

[38] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. "Recovering Architectural Design Decisions." In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 95–9509.

[39] Arman Shahbazian, Daye Nam, and Nenad Medvidovic. "Toward Predicting Architectural Significance of Implementation Issues." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Association for Computing Machinery, 2018, 215–219. ISBN: 9781450357166. DOI: 10.1145/3196398.3196440. URL: https://doi.org/10.1145/3196398.3196440.

[40] 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.

[41] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN: 0470167742.

[42] Byron J. Williams and Jeffrey C. Carver. "Characterizing Software Architecture Changes: A Systematic Review." In: *Inf. Softw. Technol.* 52.1 (Jan. 2010), 31–51. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.07.002. URL: https://doi.org/10.1016/j.infsof.2009.07.002.

[43] Robert K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Fourth Edition. Sage Publications, 2008. ISBN: 1412960991. URL: http://www.amazon.de/Case-Study-Research-Methods-Applied/dp/1412960991%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D1412960991.