The Journal of Systems and Software 217 (2024) 112170

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells☆

Philipp Gnoyke [a,b,*], Sandro Schulze [c], Jacob Krüger [d]

[a] Otto-von-Guericke University Magdeburg, Magdeburg, Germany
[b] KSB SE & Co. KGaA, Pegnitz, Germany
[c] Anhalt University of Applied Sciences, Köthen, Germany
[d] Eindhoven University of Technology, Eindhoven, The Netherlands

ARTICLE INFO

Keywords:
Software quality
Technical debt
Architecture smells
Cyclic dependencies
Empirical study
Evolutionary analysis

ABSTRACT

Architecture smells are a widely established concept to describe symptoms of software degradation by measuring perceived violations of software-design principles. As such, architecture smells can help developers assess and understand the architectural quality of their software system. However, research has rarely been concerned with how architecture smells evolve and whether they actually foster software degradation during a system's evolution. Building on our previous work in this direction, we present extended techniques for measuring architecture smells, novel visualizations, as well as an empirical study of how architecture smells evolve and what typical patterns they exhibit in 485 releases of 14 open-source systems. Among others, the results of our study indicate that especially cyclic dependencies on the class-level are prone to becoming highly complex over time, with one of the reasons being the continued merging of smells, most often resulting in tangled multi-hubs. Moreover, we found unstable dependencies to mostly grow slowly over time, whereas hub-like dependencies remain rather stable during a system's evolution. These findings are valuable for practitioners to identify and tackle system degeneration, as well as for researchers to scope new research on managing architecture smells and technical debt.

1. Introduction

Long-living software systems are often degrading over time (also known as aging or decaying), meaning that the changes to a system are likely to worsen that system's design, architecture, or structure, thus reducing its internal quality (Belady and Lehman, 1976; Parnas, 1994; Izurieta and Bieman, 2007; Ahmed et al., 2015; ISO/IEC 25010:2011(E), 2011). Consequently, evolving a software system through new features, bug fixes, or redesigns can make that system less maintainable and reliable, resulting in future changes becoming more tedious and time-consuming (Zazworka et al., 2011; Olbrich et al., 2009; Khomh et al., 2009; Besker et al., 2018, 2019). To facilitate the management of software degradation, researchers and practitioners have proposed various concepts, most prominently *code smells*, *architecture smells*, and *technical debt*. Code smells (Fowler, 2019; Lacerda et al., 2020) and architecture smells (Garcia et al., 2009b,a) are intended to describe symptoms of software degradation, helping to locate perceived negative effects in a system and enabling countermeasures, such as refactoring (Fowler, 2019; Lippert and Roock, 2006; Lacerda et al., 2020). Technical debt has recently gained more and more attention as a metaphor in the context of software degradation in which the symptoms of degradation (e.g., smells) are considered as an interest rate of postponing design decisions or software improvements (Cunningham, 1992; Kruchten et al., 2012; Li et al., 2015).

Code smells and technical debt have been researched extensively regarding their longevity, impact on maintenance efforts, or usefulness to represent software degradation (Lacerda et al., 2020; Santos et al., 2018; Cairo et al., 2018; Zhang et al., 2011; Besker et al., 2018, 2019). Despite this research, the empirical evidence regarding the relevance of code smells for software degradation is not fully conclusive (Santos et al., 2018; Zhang et al., 2011; Sjøberg et al., 2012). Similarly, researchers have studied detection techniques, the impact, and refactoring of architecture smells (Garcia et al., 2009b,a; Arcelli Fontana et al., 2017; Azadi et al., 2019; Rizzi et al., 2018), but little research has studied their evolution and impact on software degradation (Sas et al., 2019, 2022b; Rangnau, 2020; Roveda et al., 2018a). To improve on this situation, we (Gnoyke et al., 2021, 2023) have previously conducted an empirical study on the evolution of architecture smells in open-source software systems. Primarily, we tackled the question to what extent and

☆ Editor: Gabriele Bavota.

* Corresponding author at: Otto-von-Guericke University Magdeburg, Magdeburg, Germany.
E-mail addresses: philipp.gnoyke@t-online.de (P. Gnoyke), sandro.schulze@hs-anhalt.de (S. Schulze), j.kruger@tue.nl (J. Krüger).

https://doi.org/10.1016/j.jss.2024.112170
Received 22 December 2023; Received in revised form 23 June 2024; Accepted 26 July 2024
Available online 7 August 2024

under what circumstances architecture smells contribute to software degradation. For this purpose, we connected the evolution of different architecture smells with technical debt as a measure for degradation, and studied which types of architecture smells have a greater impact on software degradation under what circumstances.

In this article, we advance on existing work by contributing sophisticated techniques for measuring the evolution of architecture smells, proposing new ones for visualizing this evolution, and using our advanced tooling to conduct a new empirical study on how architecture smells themselves evolve. Consequently, instead of studying the impact of architecture smells on system degradation as in our previous work (Gnoyke et al., 2021), we are researching the architecture smells themselves in this article. Moreover, while some research questions such as the evolution of smell properties bear similarity to related work (Sas et al., 2022b), we study aspects of architecture smell evolution from different perspectives and with added context, for example by looking at different quantiles of properties and the age of smells. Throughout this study, we link the (inter-)related concepts of architecture smells and technical debt. This provides a more holistic and therefore accurate view of software evolution and system degradation. Smells (and specifically architecture smells) are often regarded as an indicator of technical debt (Suryanarayana et al., 2014; Zazworka et al., 2014; Martini et al., 2018; Nayebi et al., 2019; Rachow and Riebisch, 2022; Das et al., 2022; Sas et al., 2022a). Studying the impact of architecture smells on software maintainability and quality without taking technical debt into account would thus be incomplete.

More precisely, we contribute the following in this article:

- We propose techniques for tracking, visualizing, and measuring architecture smells, with which we advance on our previous work (Gnoyke et al., 2021). Specifically, neither our previous work nor related work included the concept of different transition types or visualizations for cyclic-dependency evolution graphs along with the underlying dependency structures. Moreover, we are generally expanding on explanations of our techniques, aiming to better convey the reasoning of how we set up this study, increase reproduceability, and spark further developments in evolutionary analyses of architecture smells.
- We report an empirical study with which we shed light into the evolution of architecture smells in real-world software systems and discuss the implications of our findings. Our research questions within this study are completely different and focus on different aspects of architecture-smell evolution compared to our previous work (Gnoyke et al., 2021). In summary, we look at different system and smell properties and we go into more detail as well as beyond descriptive statistics. Specifically, while we had already established cyclic-dependency evolution graphs, merges, and splits, we merely used them for determining smell introductions, removals, and lifespans. Now, we are fully studying the patterns and implications of complex cyclic dependency families, as well as the impact of merges and splits on aspects like shape changes. Furthermore, we previously investigated how the values of each smell property related to its remaining lifespan. In this study, we are instead observing how properties change with increasing smell ages, including their overall range and extreme values. Lastly, we previously studied the evolution of technical debt via descriptive statistics. Now, we are specifically looking for compounding trends in different quantitative metrics using statistical tests.
- We share our techniques as open-source software and the study results in a persistent repository.[1] This repository contains our previous work, our toolchain including the employed queries, additional charts, and a readme that details how to use the repository.

[1] https://zenodo.org/records/12507338

The results of our study provide new insights into how architecture smells evolve within a software system, indicating that some types are particularly cumbersome to deal with. So, our contributions can help practitioners identify, understand, and address system degradation, and provide a foundation for researchers to build upon in terms of techniques, as well as open research directions.

The remainder of this article is structured as follows. We introduce the general background on software quality and architecture smells in Section 2. Then, we describe our techniques for tracking, visualizing, and measuring architecture smells in Section 3. In Section 4, we specify the design of our study, the results of which we present and discuss in Section 5. Afterwards, we summarize the related work in Section 6 and the threats to validity in Section 7, before concluding this article in Section 8.

2. Background

In this section, we summarize the general background on software quality, architecture smells, and technical debt. Our aim is to define the central technical terms, explain the relevance and major properties of each subject, and how it relates to the other subjects. Specifically, we consider software quality as the overarching motivation of studying smells and technical debt. While smells are our main research focus in this article, we create a link to technical debt throughout it, as a focus on both of these related subjects provides a more holistic view on software quality (cf. Section 1).

2.1. Software quality and architecture

According to the ISO/IEC 25010:2011(E) (2011), the quality of a (software) system refers to the extent to which that system provides value by fulfilling its stakeholders' specified and implied needs. This definition highlights that quality assurance is essential to ensure that investments into software development and maintenance provide a benefit—in the short and long run (Naik and Tripathy, 2008). The *product quality model* defines a set of factors that contribute to software quality, which are: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability (ISO/IEC 25010:2011(E), 2011). While some of these factors are rather discernible, straightforward to assess after a release, and usually explicitly stated by stakeholders; maintainability can easily be overlooked and is harder to grasp, as it contributes to *internal quality*. Typically, a lack of maintainability is not immediately visible to stakeholders, and impairs software quality only after a longer period of time. The ISO/IEC 25010:2011(E) (2011) defines maintainability as the extent to which a system can be effectively and efficiently modified by its intended maintainers. Thus, a system is maintainable if it is "easy" to understand, change, extend, reuse, and test (Bass et al., 2013; Lippert and Roock, 2006; Suryanarayana et al., 2014).

Decades of software-engineering practice and research have resulted in best practices for ensuring software quality. These can be formalized as *design principles* and *design rules*, which involve guidelines for structuring code on various levels of abstraction, often targeting a high maintainability (Martin, 2000). The *software architecture* represents the highest level of abstraction and is defined by Taylor et al. (2010) as the set of principles for deciding on the design of a system during its development and evolution. For instance, such design decisions include how the system is structured into modules (or components) and how these interact with each other. In object-oriented systems, components can constitute classes, packages, subsystems, or layers, while interactions between them can be function calls, classes having an attribute whose type is a different class, or inheritance, among others (Bass et al., 2013). We summarize all of these interactions as *dependencies*. A well-designed, non-monolithic architecture is typically perceived to ensure reusability and extensibility—to which the principle of modularity refers (Meyer, 1997).

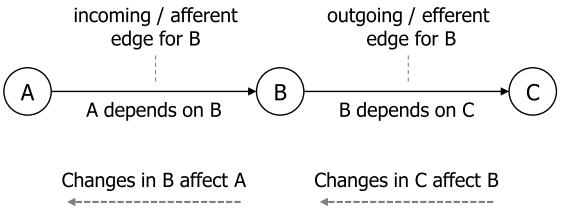

Fig. 1. Concept of dependencies between components and the effect of changes as a graph.

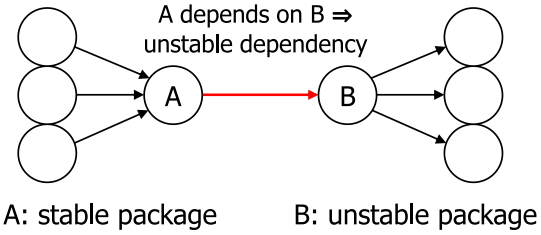

Fig. 2. A conceptual example for an unstable dependency based on Martin (2000).

When designing a software architecture, a major goal is usually to ensure a high *cohesion* within the same component and a low *coupling* between different components (Bass et al., 2013; Lippert and Roock, 2006; Stevens et al., 1974). In other words, a component should, on the one hand, have a single concern, with its elements being closely related to one another (*single responsibility principle*). On the other hand, there should only be few connections (i.e., dependencies) with little data flow to other components. This is also expressed via design rules like *few interfaces* and *small interfaces* (Meyer, 1997). Coupling should be reduced to avoid that changes to one component force adaptations to other components that depend on it (Gorton, 2011; Lippert and Roock, 2006; Suryanarayana et al., 2014). We visualize this idea in Fig. 1 with components as vertices and dependencies as edges, which is also how we depict dependency structures of architecture smells in this article.

If a software architecture violates design principles and rules, changes to components can induce disproportionate efforts due to the necessary updates to other components; potentially requiring multiple iterations. This is referred to as *ripple effects* (Gorton, 2011). To avoid such problems and facilitate maintenance, several design principles have been proposed. For example, according to the *acyclic dependency principle*, components should depend on each other in tree-like structures, while circular references should be avoided. The *stable dependency principle* demands that components should only depend on components that are at least as stable (i.e., not prone to changes, cf. Section 2.2) as they are themselves (Lippert and Roock, 2006).

2.2. Architecture smells

The term *smell* was originally coined as part of the term *code smell*, which according to Fowler (2006) is an indicator at the surface that hints at deeper problems within a system. In other words, smells are symptoms of violated design principles and rules (Azadi et al., 2019). They indicate the need for *refactoring*, which aims to improve code for maintainability without altering user-perceived functionality (Fowler, 2019). While evaluating a system's conformity with design principles and rules per se can be complex, it is usually straightforward to detect smells manually or via automated tooling (Fowler, 2006; Azadi et al., 2019; Sharma et al., 2020) like `Arcan` (Arcelli Fontana et al., 2017), which we employ (cf. Section 4.3). Not every detected smell represents an issue that has to be resolved (Olbrich et al., 2010), but smells in themselves are often perceived to be deteriorating software quality, fostering change propagation, risking ripple effects, and having negative effects on compiling and runtime performance (Martini et al., 2018; Arcelli Fontana et al., 2023; Lippert and Roock, 2006; Sas et al., 2022a).

We can classify smells according to the level of abstraction they impact. These levels range from fine-granular issues within methods/functions and classes, which are referred to as code smells, to macro-level issues concerning the architectural design of systems, referred to as *architecture smells*. While some studies found that the presence of code smells can also correlate with architectural decay (Macia et al., 2012; Vidal et al., 2016), Arcelli Fontana et al. (2019) have shown in a large-scale study that, in most cases, the presence of architecture smells cannot be predicted from code smells. Therefore, we need to consider architecture smells separately. Over time, different types of architecture smells have been identified and categorized (Lippert and Roock, 2006; Garcia et al., 2009b; Mo et al., 2015; Le and Medvidovic, 2016; Azadi et al., 2019). Frequently researched types that can be reliably detected by open-source tools include *cyclic dependencies*, *hub-like dependencies*, and *unstable dependencies* (Melton and Tempero, 2007; Arcelli Fontana et al., 2016a; Avgeriou et al., 2016; Roveda, 2018; Díaz-Pace et al., 2018; Martini et al., 2018; Sas et al., 2019; Herold, 2020; Sas et al., 2022b). We detail each of these architecture smell types in the following.

Unstable Dependencies. An unstable dependency is a component that depends on less stable components, and thus violates the stable dependency principle (Arcelli Fontana et al., 2016a). Depending on the exact definition, a component is referred to as *stable* if it did not change frequently in the past or if it can be expected to not change frequently in the future (Mo et al., 2015; Azadi et al., 2019). Because changes may require further propagation to other components, an otherwise stable component can be forced to change due to its unstable dependencies. Such cases can then increase the maintenance effort. Martin (2000, 1994) approximated a component's stability based on the effort it would need to perform a change. This approximation builds on the hypothesis that developers avoid changing components that many other components depend on, since this could lead to ripple effects and thus a higher workload. We can compute a component's instability as follows:

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

This metric builds on the numbers of a component's incoming dependencies (i.e., *afferent coupling* C_a) as well as outgoing dependencies (i.e., *efferent coupling* C_e). We visualize an example for this metric in Fig. 2, where component A has an instability value of 0.25, while it depends on the more unstable component B with an instability of 0.75. As `Arcan` is using Martin's instability metric, it also represents the basis for unstable dependencies in this study.

In this article, we regard a component that depends on multiple less stable components as a single instance of an unstable-dependency smell. Furthermore, we follow Martin (2000) and `Arcan` by focusing on unstable dependencies on the *package-level*, as opposed to the *class-level*. Specifically, by following the former we only consider packages that depend on less stable packages as unstable dependencies, whereas dependencies between classes would be considered for the latter.

Hub-Like Dependencies. A hub-like dependency refers to a component with a high number of both incoming and outgoing dependencies (i.e., high *fan-in* and *fan-out* values). Hub-like dependencies are often detected on the class-level, which we adhere to in our work. The central component can thus be referred to as the *hub-class*. As we show in Fig. 3, a hub-class is usually overloaded with responsibilities and resembles a bottleneck in the dependency structure. This increases coupling, reduces modularity, and violates the few-interfaces rule. More precisely, a high number of incoming dependencies easily leads to

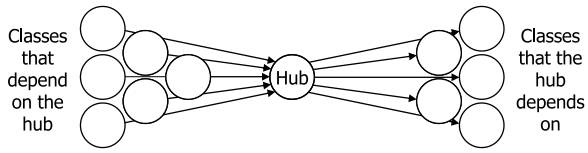

Fig. 3. A conceptual example for a hub-like dependency based on Roveda (2018).

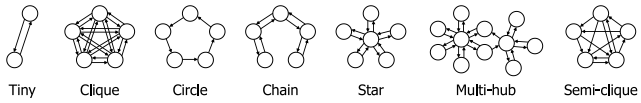

Fig. 4. The different topologies for cyclic dependencies based on Al-Mutawa et al. (2014).

many changes being propagated to the hub, which in turn risks a high number of necessary adaptations in the (many) classes that depend on the hub. Such ripple effects of hubs can multiply the number of changes required in a system (Meyer, 1997; Suryanarayana et al., 2014; Arcelli Fontana et al., 2016a; Azadi et al., 2019).

The exact criteria for detecting hub-like dependencies vary between tools (Azadi et al., 2019). In `Arcan` (Arcelli Fontana et al., 2017), hub-like dependencies are defined as having both more incoming and outgoing dependencies than the median class in the subject system. Furthermore, the hub must be *balanced*, meaning that the difference between incoming and outgoing dependencies must not be greater than $\frac{1}{4}$ of their sum. Through a case study on the effects of different types of architecture smells, Martini et al. (2018) found that practitioners tend to experience negative impacts of hub-like dependencies most frequently.

Cyclic Dependencies. A cyclic dependency constitutes a series of dependencies between components that results in at least two components reciprocally depending on each other; either directly or indirectly via other components (Melton and Tempero, 2007; Azadi et al., 2019). As we outline in more detail in Section 3.2, some tools and studies refer to *strongly connected* (i.e., potentially complex tangles with various dependencies between them) components as cyclic dependencies (Tarjan, 1972; Al-Mutawa et al., 2014). In contrast, cyclic dependencies are sometimes regarded as *simple cyclic paths* through a system's dependency structure (Sedgewick and Wayne, 2011; Roveda, 2018). We build on the former definition. Cyclic dependencies violate the acyclic dependency principle, increase coupling, and undermine modularity. For instance, components that are affected by a cyclic dependency can often not be maintained and deployed separately, because changes ripple through the cycle, potentially in multiple iterations (Martin, 2000; Azadi et al., 2019; Lippert and Roock, 2006; Suryanarayana et al., 2014). A cyclic dependency can be detected on the class-level and package-level, both of which we investigate in our study. The latter is often considered to be more critical (Sangwan et al., 2008; Azadi et al., 2019). Martini et al. (2018) found that practitioners perceive cyclic dependencies to affect quality, their speed, and the occurrence of bugs the most among the studied architecture smell types.

Al-Mutawa et al. (2014) have introduced a distinction of cyclic dependencies according to their *shape*, which they refer to as *topology*. Specifically, they classify cyclic dependencies into seven different shapes with varying levels of complexity. We display conceptual examples for the seven types in Fig. 4. If a cyclic dependency cannot be assigned to any of the depicted shapes, it is classified as *unknown*.

2.3. Technical debt

While architecture smells are a way of identifying violated design principles and rules, subpar design decisions and quality issues in a system are often summarized as *technical debt*. The metaphor of technical debt was first introduced by Cunningham (1992) and reflects financial debt in multiple ways: technical debt can be taken on *inadvertently* or *deliberately*, for example, to reduce the time to market by disregarding the originally defined architectural layout or to fix an urgent security breach. Moreover, the manner of accruing technical debt can be *reckless* or *prudent*, meaning that developers may take the consequences of reduced maintainability into account. Said maintainability reduction represents the interest rate, as any further development becomes less efficient, with the risk of *technical bankruptcy* in case the code becomes unmanageable. Ultimately, developers can repay technical debt by investing time and effort into refactoring the system. Neglecting refactoring may result in the uncontrolled growth of technical debt, which causes the system to erode and can, for example, be seen in the spread and growth of architecture smells (Cunningham, 1992; Fowler, 2009; Avgeriou et al., 2016; Suryanarayana et al., 2014; Allman, 2012). Various different problems in a system can be expressed within the metaphor of technical debt. This includes unclean code, violated design principles and rules, architectural violations, low-quality documentation and test coverage, as well as — in some definitions — the presence of bugs and non-conformity to requirements (Kruchten et al., 2012; Power, 2013; Alves et al., 2016; Suryanarayana et al., 2014). In our study, we focus on architectural technical debt.

To better estimate and analyze the presence and impact of technical debt, various quantification methods have been proposed. While outright estimating how many person-hours would be required to resolve every issue is not a straightforward task, many tools provide an abstract approximation, which is often referred to as a *technical debt index* (Arcelli Fontana et al., 2016b; Xiao et al., 2016; von Zitzewitz, 2019). Out of the various attempts for determining a system's technical debt index based on different input parameters (Letouzey, 2012; Curtis et al., 2012; Verdecchia et al., 2018; Amanatidis et al., 2020; Sas and Avgeriou, 2023), we selected a fully-documented open-source approach that integrates well with our remaining toolchain. Precisely, since we chose `Arcan` as our tool for detecting architecture smells, we identified the approach by Roveda et al. (2018b) to match our requirements. This approach measures the severity of architecture smells in a system to calculate its technical debt index. Relating (architectural) technical debt to smells ties in with a series of related work (Suryanarayana et al., 2014; Nayebi et al., 2019; Das et al., 2022; Sas and Avgeriou, 2023). Specifically, in the approach of Roveda et al., each architecture smell is assigned a *severity score*, which is a normalized index in the range $[0, 1]$. A dataset of 109 *Qualitas Corpus* (Tempero et al., 2010) projects served as reference, with every type of architecture smell being considered separately. For instance, for a cyclic dependency, its number of affected components (i.e., vertices in the dependency graph, which we refer to as *order*) is considered. Then, this number is compared to the reference data and a quantile value from 0 to 1 is determined. The resulting severity score is then multiplied with the smell's centrality (using *PageRank*) and weighed by its order. Thus, the impact of every architecture smell can be described via a respective technical debt index. For the whole system, the sum of every smell's technical debt index represents the overall technical debt. Roveda et al. further multiply every architecture smell with its *trend evolution score*, which is set to two for growing smells, one for stagnating smells, and 0.5 for shrinking smells (Roveda, 2018; Roveda et al., 2018b). We accounted for the concerns of Sas et al. (2019) in our study by excluding this score, since it distorts the results. In Section 3.4, we specify in detail how we quantify architecture smells and technical debt for our study.

3. Concepts for analyzing architecture smells

The evolution of architecture smells has not been researched extensively (cf. Section 6.1). Particularly, existing techniques for tracking architecture smells throughout the evolution of a software system are limited in their scope and precision. For this reason, we have previously extended existing techniques for tracking architecture smells

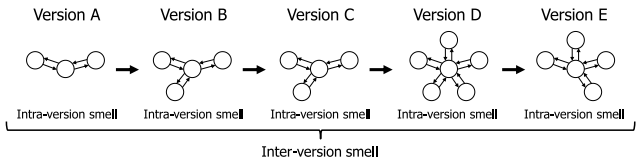

Fig. 5. Conceptual example for an inter-version cyclic-dependency smell that consists of five intra-version smells.

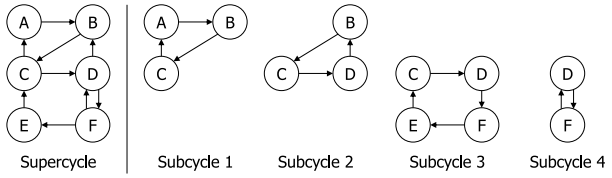

Fig. 6. Conceptual example of a supercycle (left) that comprises four subcycles (right).

to describe their evolution and measure their properties more precisely (Gnoyke et al., 2021, 2023). In this section, we describe the terminology we use (i.e., intra- and inter-version smells), our tracking and measuring techniques, and introduce feasible visualizations as well as transition types (cf. Section 3.3), which are both a completely new contribution compared to our previous work.

3.1. Intra-version and inter-version smells

We (Gnoyke et al., 2021) have established a distinction between intra-version and inter-version architecture smells, referring to

- **intra-version smells** if we consider the architecture smells that exist in a single version of a system, and
- **inter-version smells** if we consider a set of intra-version smells that are *related* and occur through multiple consecutive versions of a system.

Specifically, since a system typically evolves gradually, when looking at an intra-version smell in a particular version, we can often identify intra-version smells in subsequent versions that affect the same components and have the same properties or change slightly over time. Thus, these intra-version smells are related and define an inter-version smell. We illustrate a conceptual example for this scenario in Fig. 5, in which one cyclic-dependency inter-version smell evolves through five consecutive versions. The smell grows from version A to B and from C to D, remains stable from B to C, and shrinks from D to E. So, each version of that system comprises an intra-version smell. Since we can identify a clear relationship between these intra-version smells, they together represent an inter-version smell.

Note that one intra-version smell is always part of exactly one inter-version smell, while one inter-version smell consists of one or more intra-version smells. If an intra-version smell has no predecessors and no successors (i.e., it is removed directly after its introduction), the corresponding inter-version smell consists of only this one intra-version smell. Not referring to both intra- and inter-version smells as just smells is helpful to avoid misunderstandings and facilitate the conceptualization of measurements that are specific to one or the other kind. For example, measurements that concern the number of affected components and their dependencies are only applicable for intra-version smells, since these can considerably change over multiple versions—which we discuss in detail in Section 3.4.

3.2. Smell tracking

To track inter-version smells along a system's evolution, we first have to identify intra-version smells in a number of consecutive system versions. Then, we can compare the intra-version smells of adjacent versions to match related ones. This idea of tracking architecture smells has been introduced by Sas et al. (2019). We have extended and improved their technique to identify and track architecture smells more reliably (Gnoyke et al., 2021, 2023). Since our goal in this article is to analyze evolution patterns, we only briefly summarize our technique and, instead, focus on our novel visualizations.

In short, our technique matches system components being affected by architecture smells in adjacent versions based on the components' name. This means that we consider two components as the same only if they share the same fully qualified name (i.e., `package.class`). As a first step, we directly match hub-like and unstable dependencies in different versions that have the same central component. For hub-like dependencies, this central component is the hub class; while it is the package that depends on less stable packages for an unstable dependency.

Since this central component may be renamed between versions, a second step is necessary to improve the tracking of smells. For this purpose, similar to Sas et al., we use a greedy algorithm that iteratively matches the most similar remaining intra-version smells if their overlap in affected system components (using Jaccard set similarity) is high enough. Specifically, we found a threshold of a Jaccard similarity of a least 0.6 to be a good trade off of false-negatives and false-positives. Consequently, we can identify and track inter-version hub-like and unstable dependencies smells consisting of a one-dimensional sequence of intra-version smells. Unfortunately, the same is not possible for cyclic dependencies because of merging and splitting, which is why we propose the concepts of sub- and supercycles as well as cyclic-dependency evolution graphs to track these.

Subcycles and Supercycles. When defining what system components are impacted by a cyclic dependency, two different types have been observed in the literature (Melton and Tempero, 2007), for which we have previously established the following terminology (Gnoyke et al., 2021):

Subcycles represent *simple* or *elementary cycles*, each of which starts and ends in the same node (i.e., component) with no repeated edges (i.e., dependencies) in-between (Sedgewick and Wayne, 2011). For instance, Sas et al. use this concept of subcycles.

Supercycles represent *strongly connected components* with any number of edges between the nodes where each node can be reached from every other node (Gross et al., 2013). For example, Al-Mutawa et al. (2014) use this concept of supercycles.

Each subcycle is part of exactly one supercycle, while a supercycle consists of at least one subcycle. In Fig. 6, we display a conceptual example with a supercycle that comprises six components (nodes A–F) and contains four subcycles. A component can be part of at most one supercycle, while it can be part of an arbitrary number of subcycles, for instance, component C in Fig. 6 is part of three subcycles.

Subcycles are usually easier to understand and visualize, since they are simpler than supercycles (Laval et al., 2012). However, supercycles, unlike subcycles, are unique in their components and provide a more stable, and thus accurate, view on the evolution of an architecture smell. Specifically, subcycles are easily broken if a single edge is removed, while supercycles are not as easily broken and can represent extensive tangles. For instance, removing the edge D → B breaks subcycle 2 in Fig. 6. While the supercycle of the remaining three subcycles (all of which are still valid) would also miss the edge D → B, it would still conform to the definition of supercycles, and thus represent the same cyclic dependency. Consequently, we track evolving cyclic dependencies based on supercycles. As the criterion for matching two intra-version cyclic dependencies in adjacent system versions (i.e., two supercycles), we check that these share at least

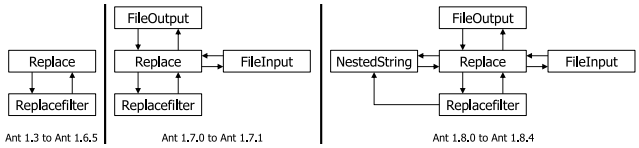

Fig. 7. Evolution of the dependencies of a one-dimensional class-level (i.e., class `Replace` and adjacent classes) cyclic dependency in the system `Ant` (package: `org.apache.tools.ant.taskdefs`).

two components. Particularly, if there exists a cyclic path between two components in both system versions, that path likely involves the same dependencies, such as class calls, and thus constitutes the same inter-version smell.

We display a real-world example for the evolution of a rather simple class-level cyclic dependency in Fig. 7. This smell is centered around the class `Replace` of the system `Ant`, and references its nested classes. The cyclic dependency gradually grows from two impacted classes to four and then to five, while the number of dependencies grows from two to six to nine. As we outline in Section 3.4, we refer to the former property as the smell's order and to the latter as its size. The number of subcycles also increases from one to three to five, ending up with:

(1) `Replace` → `Replacefilter` → `Replace`
(2) `Replace` → `FileInput` → `Replace`
(3) `Replace` → `FileOutput` → `Replace`
(4) `Replace` → `NestedString` → `Replace`
(5) `Replace` → `Replacefilter` → `NestedString` → `Replace`.

The smell's shape evolves from tiny to star (cf. Section 2.2).

When visualizing each intra-version smell as a node, we can construct the graph in Fig. 8. This graph shows the evolution of the resulting inter-version smell. We display each intra-version smell's order, size, and number of subcycles as a triple below each node in the graph, as well as its version within the analyzed time span. The graph is a trivial case of a *cyclic-dependency evolution graph*, which we describe in more detail in Section 3.3.

Merging and Splitting. While the cyclic dependency in Fig. 8 evolves in a one-dimensional way, this is not the case for all cyclic dependencies. Adding one or more dependencies among system components can lead to the *merging* of two or more intra-version cyclic dependencies. Similarly, removing one or more dependencies can cause the *splitting* of an intra-version cyclic dependency into multiple ones.

We display a real-world example for this phenomenon from the system `Lucene` in Fig. 9. Specifically, we show two class-level cyclic dependencies that merge into a single one. Before the merge, there is a circle cyclic dependency around the class `MMapDirectory`, which references its nested classes, plus a chain cyclic dependency around the class `FSDirectory`, which references its inherited classes. The class `MMapDirectory` references the class `FSDirectory`, but not the other way around, which is why the two intra-version cyclic dependencies are not a single one from the beginning. Eventually, a dependency from `FSDirectory` to `MMapDirectory` is introduced, which is why all involved components become reachable from every other component. If we reverse the chronological order of this example (i.e., reading Fig. 9 from right to left), it would represent the splitting of one cyclic dependency into two.

To correctly track cyclic dependencies, we implemented our technique to take merging and splitting into account. As we outline with empirical data in Section 5.1, only by using this tracking concept, we can analyze why particular intra-version cyclic dependencies span over wide parts of a single version and why particular inter-version cyclic dependencies span over extensive parts of a system's evolution. The resulting complex instances of smells considerably contribute to the degeneration of software quality.

3.3. Visualizing cyclic dependencies

We introduced parts of the above concepts for tracking the evolution of architecture smells in our previous work (Gnoyke et al., 2021). However, we did not propose visualizations for the evolution of architecture smells. Next, we describe our novel contribution of using evolution graphs for visualizing how architecture smells evolve throughout a system's versions.

While the evolution of hub-like and unstable dependencies can be displayed using a one-dimensional graph, this is not sufficient for inter-version cyclic dependencies. For example, considering Fig. 9, a one-dimensional graph does not allow to properly visualize code restructurings that involve splitting and merging of the intra-version smells. This is why we introduce the notion of *cyclic-dependency evolution graphs*: to track the evolution of cyclic dependencies, we create a system-wide graph of all intra-version cyclic dependencies. More precisely, all intra-version cyclic dependencies of the same version are part of the same layer. Then, we represent relationships using directed edges pointing towards the successive version. We refer to these edges as *transitions*. In the end, we can display the constructed evolution graphs using the well-suited *Sugiyama graphs* (Sugiyama et al., 1981).

We display a conceptual example for such a system-wide cyclic-dependency evolution graph in Fig. 10. Note that the colors in Fig. 10 constitute a visual aid. They represent neither the severity of the cyclic dependencies nor any degree of relationship. The system consists of the nine components A to I that undergo changes in their dependency structure in the seven depicted versions. In total, 16 intra-version cyclic dependencies exist (bottom). The system-wide cyclic-dependency evolution graph is not connected (top). Instead, the evolution graph involves three connected sub-graphs (families), each constituting an inter-version cyclic dependency. The smallest of these three (family 2) consists of only a single intra-version cyclic dependency, specifically B3. In the largest sub-graph (family 1), 12 intra-version cyclic dependencies and multiple splits as well as merges are involved. We refer to an inter-version cyclic dependency as a *family*, because it can encompass multiple intra-version cyclic dependencies in the same version. Each cyclic-dependency family can be represented as an individual, connected evolution graph. We furthermore use the term *branch* to refer to a subset of a cyclic-dependency evolution graph that forms a directed path graph—a sequence of intra-version smells that evolve without merging or splitting (Gross et al., 2013). For instance, A1→B1→C1 as well as B2→C2 form a branch.

We classify transitions into three types. A transition from the intra-version smell A to the intra-version smell B is a

- *pure transition* if B is the only successor of A, and A is the only predecessor of B.
- *merge transition* if B has more than one predecessor.
- *split transition* if A has more than one successor.

A transition can be both, a merge and a split, at the same time. For instance, the transitions in Fig. 10 are a

- pure transition: A1→B1, B1→C1, B2→C2, F2→G1, E3→F3, F3→G2
- merge transition: C1→D2, C2→D2, D1→E1, D2→E1, E1→F2, E2→F2
- split transition: D2→E1, D2→E2, E1→F1, E1→F2

Note that D2→E1 and E1→F2 are split and merge transitions at the same time.

We show the cyclic-dependency evolution graph for the real-world example of Fig. 9 in Fig. 11. Before and after the depicted merging from version 18 (`Lucene 3.0.3`) to 19 (`Lucene 3.1.0`), changes to the set of affected components and dependencies occur. This eventually results in an order of ten, size of 20, and nine sub-cycles. We show another example in Fig. 12, which represents a package-level cyclic-dependency family from the system `Jung` with two merges and one split.

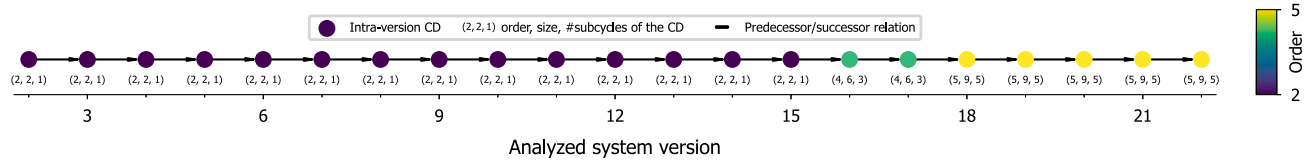

Fig. 8. Example of an evolution graph for the one-dimensional class-level cyclic dependency (CD) from Fig. 7.

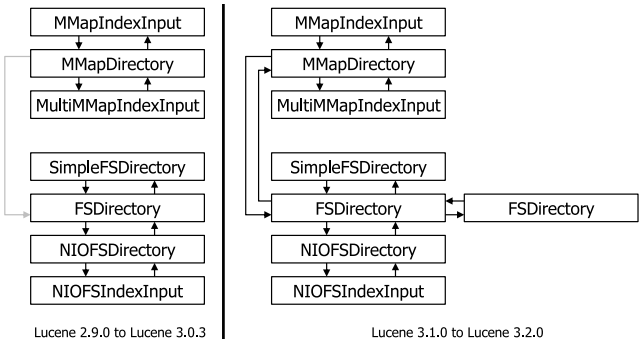

Fig. 9. Merging (left to right) of two cyclic dependencies in the system `Lucene` around the classes `MMapDirectory` and `FSDirectory` (package: `org.apache.lucene.store`). Reversing the evolution (i.e., from right to left) represents the splitting of one cyclic dependency.

3.4. Smell properties

To describe architecture smells on a quantitative basis, we have implemented our tooling to measure metrics for a variety of smell properties. We differentiate between properties of intra-version smells, inter-version smells, and system versions. In the following, we explain the intra-version and inter-version smell properties that we employ in our study, providing more details compared to our previous work (Gnoyke et al., 2021). We display an overview of these properties in Table 1. As we can see in the last column (references), more than half of these properties have been used in previous studies.

Intra-Version Smell Properties. Exact values for intra-version smell properties can only be determined for an individual intra-version smell. These properties inform us on how a smell affects the system in the respective version. Values of related intra-version smells can be analyzed as a time-series to understand how a smell's severity evolves over different versions of a system.

The **general properties** (listed as *all* in the column *smells*) can be used to describe intra-version instances of any architecture smell type. From Sas et al. (2019), we have adopted the notion of a smell's *age*. We define this age as the number of versions that have passed since the intra-version smell's first predecessor has been introduced, equaling 0 if there are no predecessors. Regarding cyclic dependencies with a non-one-dimensional evolution like in Fig. 11, we backtrack in the cyclic-dependency evolution graph. For instance, the intra-version smell's age in version 30 equals 18, as its oldest predecessor was introduced in version 12. When looking at the intra-version smell with three affected classes in version 18, its oldest predecessor was introduced in version 14, resulting in an age of 4 versions. On a similar note, we define an intra-version smell's *remaining age* as the number of versions that are left before its last successor is removed. This property equals 1 when an intra-version smell has only a single successor and 0 if it does not have any successors. It can only be determined retrospectively, when the history of a system is analyzed. The age and remaining age are our only intra-version smell properties that require information about the smell's relation to intra-version smells in other versions.

Following graph theory, we refer to the number of a smell's affected components as its *order* and to the number of dependencies among them as its *size* (Gross et al., 2013). For example, the supercycle cyclic dependency in Fig. 6 has an order of 6 and a size of 9. We relate these two properties in the *size overcomplexity*. This property measures the share of edges that could be removed without reducing the number of affected components. In the case of cyclic dependencies, this means that every vertex in the cyclic dependency remains part of the strongly connected component despite the removed edges. High size overcomplexities indicate tangled and complex smells with many paths between their components. We calculate the corresponding metric with the formula:

$$size\ overcomplexity = \frac{size\ -\ minimum\ size}{size} \tag{2}$$

Minimum size represents the smallest number of edges between the affected components that are necessary to affect all of them. To simplify the computation, we do not perform graph traversals to determine the *minimum edge set*, but, instead, assume the following: for cyclic dependencies, the minimum size equals their order, which is the case for tiny and circle shaped cyclic dependencies, the least complex cyclic dependencies. Consequently, the size overcomplexity for the example in Fig. 6 equals $\frac{1}{3}$. For hub-like and unstable dependencies, the minimum size is: $order-1$. That is because, for both, all affected components must at least be connected with one edge to the central component.

We also define the properties *centrality* and *overlap ratio* based on the work of Sas et al., which put a smell's position into the context of other components and smells. The former represents the smell's *PageRank* value in the system's dependency structure. Higher values indicate that the affected components are more central, meaning that these are referenced by many components or other central components. The overlap ratio equals 0 if the smell does not overlap at all with other smells. It equals 1 if the smell's affected components are all affected by at least one other smell. Given that we concentrate on supercycles when analyzing cyclic dependencies (cf. Section 3.2), we mostly observe lower overlap ratios for cyclic dependencies than Sas et al. did. Central and overlapping smells tend to be harder to refactor and more severe, as they are more interconnected within the system (Sas et al., 2019).

To relate architecture smells with *technical debt*, we adapt a quantification proposed by Roveda (2018), as described in Section 2.3. This quantification assigns a *severity score* to every intra-version smell, representing a normalized value between 0 and 1 with higher values indicating more severity. Simplified, the severity score is obtained by determining the smell's quantile (i.e., cumulative probability) when comparing its order to a reference dataset from the *Qualitas Corpus*. Together with the smell's order and its centrality, we can determine an individual technical debt value. The sum of all smells' technical-debt values constitutes the version's technical-debt index. By assigning an individual technical-debt value to each smell, smells can be identified for refactoring in a goal-oriented way.

Other properties are **specific to cyclic dependencies** (listed as *CD* in the column *smells*). These include the *shape*, which indicates a smell's complexity. The shape assignment algorithm by Al-Mutawa et al. (2014) classifies each cyclic dependency as one of the shapes in Fig. 4 or as *unknown*. As we outlined in Section 3.2, the *number of subcycles* can be another indicator of a cyclic-dependency's severity. Following Laval et al. (2012) and Sas et al. (2019), we also employ the *number of inheritance edges* as a property of cyclic dependencies. Inheritance edges are a subset of the edges in the dependency graph of a smell's affected components. They can be an indicator of intentional

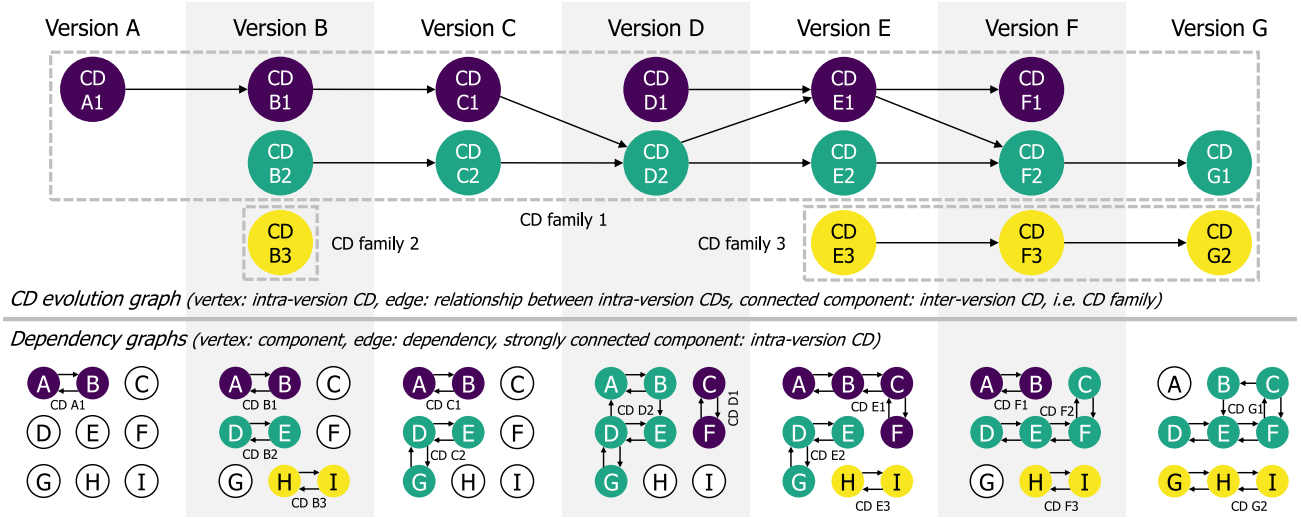

Fig. 10. Conceptual example of a system-wide evolution graph of cyclic dependencies (CDs).

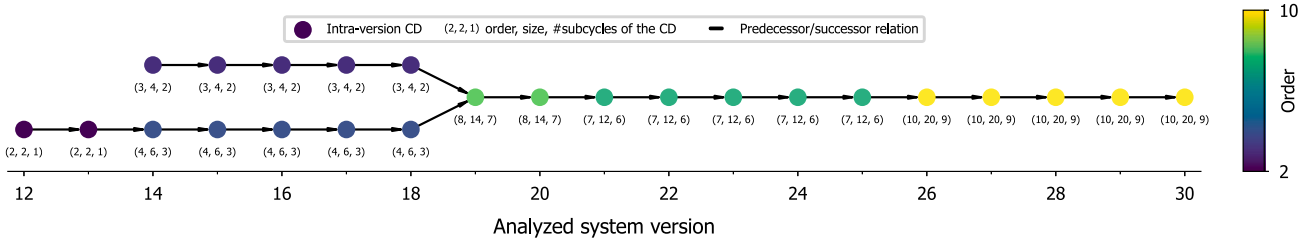

Fig. 11. Example of a class-level cyclic-dependency (CD) evolution graph with merging in the system `Lucene` around the classes `MMapDirectory` and `FSDirectory` (cf. Fig. 9).

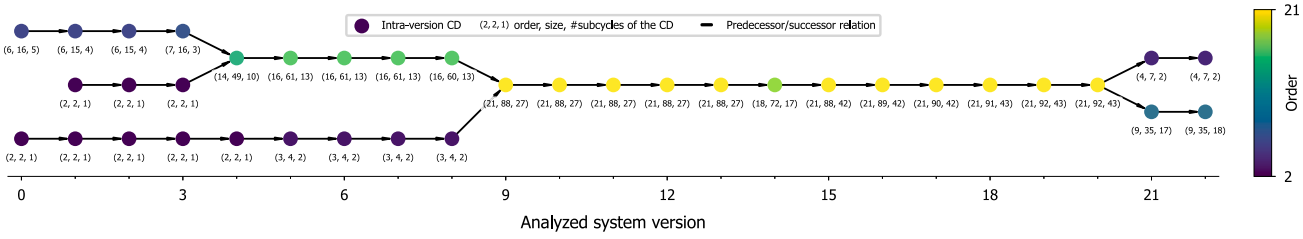

Fig. 12. Example of a package-level cyclic-dependency (CD) evolution graph with mergings and a split in the system Jung.

design, whereas other references like method invocations are more likely to unintentionally cause a cycle. We only consider inheritance relationships in class-level cyclic dependencies, since an edge in a package-level cyclic dependency can represent several dependencies between various pairs of classes. To provide a normalized value, we furthermore compute the *relative number of inheritance edges* in relation to the size (i.e., the number of all edges in a cyclic dependency).

Two major properties **specific to hub-like dependencies** (listed as *HD* in the column *smells*) are the number of incoming edges into the hub class and the number of outgoing edges from the hub class. We refer to the former as the *afferent coupling* C_a and to the latter as the *efferent coupling* C_e. Using the *hub ratio*, we relate these two properties as:

$$hub\ ratio = \frac{C_a - C_e}{C_a + C_e} \tag{3}$$

The hub ratio is positive for hubs with more incoming than outgoing dependencies and negative for the other way around. As we use `Arcan` to detect smells (cf. Section 4.3), a hub-like dependency is only detected if its difference between the numbers of incoming and outgoing dependencies is not higher than a quarter of their sum (Arcelli Fontana et al., 2016a). Thus, the hub ratio ranges from $-\frac{1}{4}$ to $+\frac{1}{4}$. For instance, the hub in Fig. 3 has an afferent coupling of 6, efferent coupling of 5, and thus a hub ratio of $\frac{1}{11}$.

Two properties describing **specifically unstable dependencies** (listed as *UD* in the column *smells*) are the *degree of unstable dependency* (DoUD) and the *instability gap*. The former has been introduced in `Arcan`. It is defined as the number of less stable components divided by the total number of components that the unstable dependency's main component depends on Arcelli Fontana et al. (2017). Meanwhile, the instability gap represents the difference in the instability value (cf. Section 2.3) between the main component and the less stable components that it depends on. While Sas et al. (2019), who introduced this property, utilize the mean value of the less stable components' instabilities, we argue that their distribution can be of interest as well. Thus, we measure both the *median* value and *inter quartile range* (IQR) of the instability gap. High values of both the DoUD and instability gap increase the risk of ripple effects to the less stable component.

Inter-Version Smell Properties. Some properties are applicable to entire inter-version smells, and cannot be analyzed as time-series. These properties consider lifetime aspects of smells and are of interest

Table 1
Properties of architecture smells that we measure in our study.

Smells	Property	Range	Description	Other names in literature	References
intra-version smell properties					
All	Age	$[0..\infty)$	Number of versions since the introduction of the smell's first predecessor		Sas et al. (2019)
All	Remaining age	$[0..\infty)$	Number of versions before the removal of the smell's last successor		–
All	Order	$[2..\infty)$	Number of affected components (vertices in the dependency graph)	size, architectural smell weight, number of vertices	Al-Mutawa et al. (2014), Roveda (2018), Sas et al. (2019)
All	Size	$[1..\infty)$	Number of dependencies between the affected components (edges in the dependency graph)	number of edges	Sas et al. (2019)
All	Size overcomplexity	$[0, 1)$	Share of edges between affected components that are not necessary to form the smell		–
All	Centrality	$[0, \infty)$	PageRank of the smell in the dependency structure (CD: highest PageRank of the affected components; HD, UD: PageRank of the central component)	PageRank	Roveda (2018), Sas et al. (2019)
All	Overlap ratio	$[0, 1]$	Share of the smell's components that are affected by at least one other smell		Sas et al. (2019)
All	Severity score	$[0, 1]$	Normalized metric to assess the smell's severity		Roveda (2018)
All	Technical debt	$[0, \infty)$	Quantified TD that the smell contributes to the TDI		Roveda (2018)
CD	Shape	n/a	One of the seven shapes in Fig. 4 or unknown	topology	Al-Mutawa et al. (2014), Sas et al. (2019)
CD	Number of subcycles	$[1..\infty)$	Number of simple cycles in the supercycle		
c.-l. CD	Number of inheritance edges	$[0..\infty)$	Number of dependencies in the supercycle that represent inheritance relationships		Laval et al. (2012), Sas et al. (2019)
c.-l. CD	Relative number of inheritance edges	$[0, 1]$	Ratio of the number of inheritance edges to the size		–
HD	Afferent coupling	$[1..\infty)$	Number of incoming edges into the hub class		Martin (1994)
HD	Efferent coupling	$[1..\infty)$	Number of outgoing edges from the hub class		Martin (1994)
HD	Hub ratio	$[-0.25, 0.25]$	Difference of the number of incoming and outgoing dependencies divided by their sum		Arcelli Fontana et al. (2016a)
UD	Degree of unstable dependency (DoUD)	$(0, 1]$	Ratio of less stable referenced packages to all packages that are referenced by the main package	strength	Roveda (2018), Sas et al. (2019)
UD	Instability gap median	$(0, 1]$	Median of the differences in instability between the main package and its referenced less stable packages		Sas et al. (2019)
UD	Instability gap inter-quartile range (IQR)	$[0, 1)$	Inter quartile range of the differences in instability between the main package and its referenced less stable packages		Sas et al. (2019)
inter-version smell properties					
All	Version of introduction	$[0..\infty)$	First version the smell is present in		–
All	Version of removal	$[1..\infty)$	First version the smell is not present in		–
All	Lifespan in versions	$[1..\infty)$	Number of versions the smell is present in		–
CD	Family order	$[1..\infty)$	Total number of intra-version smells in the CD family		–
CD	Family width	$[1..\infty)$	Number of co-existing intra-version smells in the same version of the CD family		–
CD	Median family width	$[1..\infty)$	Median of the family width over all versions of the CD family		–
CD	Maximum family width	$[1..\infty)$	Maximum of the family width over all versions of the CD family		–

CD: cyclic dependency; HD: hub-like dependency; UD: unstable dependency; TD: technical debt; c.-l.: class-level

from an evolutionary point of view. They provide a more in-depth understanding of when smells are introduced, changed, or removed.

For all inter-version smells, we define lifetime-oriented properties, such as the *version of introduction* and the *version of removal*. We regard the first version in which an inter-version smell cannot be observed anymore as its version of removal. The difference between these two values constitutes the smell's *lifespan*, which equals the number of intra-version smells for hub-like dependencies, unstable dependencies, and cyclic dependencies that evolved without splitting or merging. In a one-dimensional inter-version smell with a lifespan of n versions, the initial intra-version smell has an age of 0, while the last intra-version smell an age of $n - 1$. For more complex cyclic-dependency families, we define the *family order* as a family's number of intra-version smells. We refer to the number of co-existing intra-version smells of the same family in the same version as the *family width*. This property is special because it is time-dependent and can change from version to version—unlike the aforementioned inter-version smell properties. For comparing the complexity of cyclic-dependency evolution graphs, we therefore use the *median* and *maximum family width*, which aggregate the family width over an entire family.

On a final note, we remark that the time series of intra-version smells can be considered as an inter-version smell property. For example, the order of the inter-version cyclic dependency in Fig. 8 grows as a function of the version number from two to four to five. Cyclic-dependency families that involve splitting or merging require an additional step of aggregating values in co-existing intra-version smells in the same version. This can be demonstrated in version two of Fig. 12, which contains three intra-version smells. Consequently, a three-value distribution exists for every intra-version smell property. The distribution can be utilized for diverse statistical analyses (cf. Section 5.1) or can be aggregated to a mean or median for time-series examinations.

4. Case study design

Next, we describe our case study, including our research questions, input dataset, and research method.

4.1. Research questions

Using our novel techniques, we aimed to study the evolution of architecture smells to improve our understanding of how these change throughout a system's versions. For this purpose, we formulated three research questions (RQs):

RQ$_1$ *What evolution patterns can we observe for cyclic dependencies?*
- **RQ$_{1.1}$** *How do merging and splitting impact the evolution of cyclic dependencies?*
- **RQ$_{1.2}$** *How does the shape of cyclic dependencies change over time?*

Supercycle cyclic dependencies often grow extensively and eventually entangle large parts of their system (Melton and Tempero, 2007). According to a case study by Martini et al. (2018), most practitioners consider cyclic dependencies as the type of architecture smell that has the most impact on software quality, is the most relevant to identify, and requires on average the most effort during refactoring. Cyclic dependencies and their impact on software quality erosion are also a heavily researched subject and detected by many smell detection tools (Melton and Tempero, 2007; Laval et al., 2012; Al-Mutawa et al., 2014; Arcelli Fontana et al., 2016a; Avgeriou et al., 2016; Roveda, 2018; Díaz-Pace et al., 2018; Martini et al., 2018; Azadi et al., 2019; Sas et al., 2019; Herold, 2020; Sas et al., 2022b). Because of this relevance, we focused on understanding the evolution of cyclic dependencies, particularly with respect to the novel concepts of merges and splits, which no other study has focused on so far. Specifically, we investigated the circumstances under which large and complex cyclic dependencies emerge and persist in systems (**RQ$_1$**), the influence of merging and splitting (**RQ$_{1.1}$**) and the shape as an indicator for complexity (**RQ$_{1.2}$**).

In the following research questions, we additionally consider hub-like dependencies and unstable dependencies. As we want to better understand how complex and severe cyclic dependencies evolve, while the other two architecture smell types do merge and split in a similar way, we excluded them from **RQ$_1$**. The remainder of the research questions focuses on evolutionary patterns of architecture smells in general, which is why more architecture smell types increase generalizability.

RQ$_2$ *How do the properties of architecture smells evolve over time?*

Another point of view for understanding how complex and hard to refactor architecture smells manifest themselves in a software system is to analyze the evolution of their properties. Doing so, we aimed to discover whether particular properties grow or shrink over time and what values can be expected with an increasing smell age (**RQ$_2$**). For instance, if architecture smell instances with extreme values require a particular time span to reach this state, we can gain valuable insights for practitioners on refactoring strategies to prevent such an extreme state.

RQ$_3$ *How do the evolution of architecture smells and technical debt relate to the compound-interest metaphor?*

Describing poor system quality with the metaphor of technical debt includes representing reduced maintainability as an interest rate. In real-world debt, interest is often accompanied by compound interest, which causes the debt to grow exponentially if it is not reimbursed. Within the context of software, we want to explore whether and how technical debt exhibits compound interest—specifically, whether reduced maintainability leads to a faster accumulation of technical debt (**RQ$_3$**). We want to clarify that we are not studying the interest of technical debt itself, which can describe various effects outside of the system design, but rather how technical debt can feedback on itself. We focus on architecture smells as an indicator of architectural technical debt (cf. Section 2.3). This research question also contributes to understanding how issues like architecture smells establish themselves and spread in software.

Overall, we contribute novel insights into the evolution and impact of architecture smells by addressing these research questions. Our results can help practitioners in managing architecture smells, and researchers in designing new techniques.

4.2. Subject systems

To answer our research questions, we conducted a multi-case study (Yin, 2018). For this purpose, we needed a dataset of well-established systems with a variety of domains, developers, number of versions, and code sizes. Such a diverse dataset increases the external validity of our findings and conclusions. We decided to focus on open-source systems to ensure reproducibility. Moreover, due to Java's continued relevance as a programming language[2] especially for open source systems (Karus and Gall, 2011), we selected a Java-based dataset. This also accounts for `Arcan` as our tool of choice for architecture-smell detection, whose open-source version analyzes Java systems (Arcelli Fontana et al., 2017). Finally, we decided to use the established *Qualitas Corpus Evolution Distribution* dataset (Tempero et al., 2010),[3] which satisfies all aforementioned requirements and has been studied extensively in related research (Sas et al., 2019; Al-Mutawa et al., 2014; Roveda et al., 2018b; Gnoyke et al., 2021). In Table 2, we provide an overview about each system in the Qualitas Corpus dataset, including their domain, number of analyzed versions, covered time span, and code size in lines of code (LOC).

In total, we analyzed 485 versions (i.e., releases) of 14 well-established systems spanning various code sizes. Each system involves between 2 and 15 years of continued development in the form of 16 to 115 versions. Within these timespans, some systems remained mostly stable regarding their code size (e.g., `JStock`, `ArgoUML`), while others grew considerably (e.g., `Hibernate` with a factor of 61). We omitted `Eclipse` from the original Qualitas Corpus dataset, as our current tool chain does not scale to its large size. Furthermore, we excluded 42 of the 527 original versions in cases where parallel development branches were included in the corpus. We selected versions from one branch in such cases, because our analysis includes time-related measures, and is thus assuming a one-dimensional progression from version to version. For example, `JGraph` 5.4.4 was released on the same day for Java 1.3 and 1.4 separately, which is why we only utilized the version for Java 1.4.

4.3. Implementation

To automatically analyze the evolution of a software system's architecture smells and technical debt, we implemented the concepts we introduced in Section 3 in a toolchain, which we illustrate in Fig. 13. For detecting architecture smells, we chose to use `Arcan` (Arcelli Fontana et al., 2016a), which is able to detect all types of architecture smells that we are interested in (cyclic dependencies, hub-like dependencies, unstable dependencies). As version 1.21 of `Arcan` is open-source,[4] we can retrace all computations and adapt their functionality if required. Related tools with a similar scope of architecture smells, such

[2] Since its first publication in 2014, the *IEEE Spectrum ranking* constantly placed Java in the top five of the most relevant programming languages—exept for 2022 constantly in the top three (Cass, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023).

[3] http://qualitascorpus.com/download/

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

Table 2
Overview of our subject systems from the Qualitas Corpus dataset (Tempero, 2013).

System	Domain	# versions		First version			Last version		
		Original	Included	ID	Date (YMD)	LOC	ID	Date (YMD)	LOC
Ant	Build system	23	23	1.1	2000–07–18	7,837	1.8.4	2012–05–23	105,007
ANTLR	Parser generator	22	22	2.4.0	1998–09–18	2,834	4.0	2013–01–22	21,919
ArgoUML	Diagram application	16	16	0.16.1	2004–09–04	106,500	0.34	2011–12–15	192,410
Azureus/Vuze	Database	63	63	2.0.8.2	2004–03–14	62,388	4.8.1.2	2012–12–17	484,739
Freecol	Videogame	32	32	0.3.0	2004–09–30	21,309	0.10.7	2013–01–07	100,748
FreeMind	Diagram application	16	16	0.0.2	2000–06–27	2,712	0.9.0	2011–02–19	50,198
Hibernate	Database	115	106	0.8.1	2001–11–30	3,555	4.2.2	2013–05–23	217,163
JGraph	Diagram application	39	37	5.4.4-java1.4	2005–03–28	10,780	5.13.0.0	2009–09–28	22,758
JMeter	Software testing	24	24	1.8.1	2003–02–03	34,170	2.9	2013–01–28	90,612
JStock	Stock trading	31	31	1.0.6	2011–03–29	43,811	1.0.7c	2013–06–20	48,842
Jung	Diagram application	23	23	1.0.0	2003–07–31	7,206	2.0.1	2010–01–25	37,989
JUnit	Software testing	24	23	2.0	1998–01–08	1,346	4.11	2012–11–16	7,428
Lucene	Text search	36	31	1.2-final	2003–09–10	6,505	4.3.0	2013–04–27	285,804
Weka	Machine learning	63	38	3.1.7	2000–02–22	57,194	3.7.9	2013–02–21	247,805

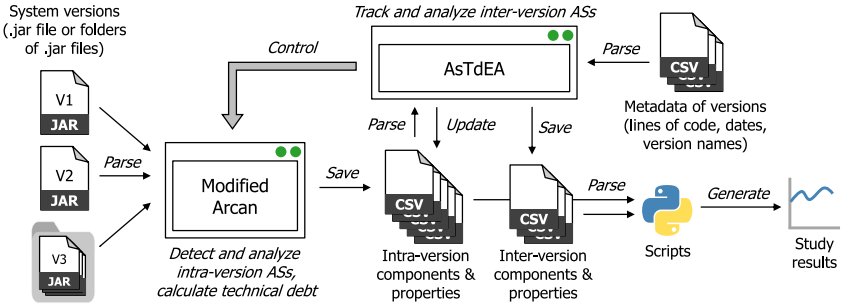

Fig. 13. Workflow of our toolchain for tracking architecture smells.

as `Designite`, did not offer this flexibility for us. Eventually, we modified `Arcan` to detect cyclic dependencies on the supercycle level in addition to the pre-implemented subcycle level (cf. Section 3.2). For this purpose, we utilized the algorithm by Tarjan (1972) for identifying strongly connected components. To determine the shape of a cyclic dependency, we also modified `Arcan` to resemble the original algorithm by Al-Mutawa et al. (2014), which was designed for supercycles. Furthermore, we added the quantification of technical debt, the computation of several properties we describe in Section 3.4, and optimized `Arcan`'s runtime, which included multi-threading and avoiding repeated graph traversals. As `Arcan` is designed to analyze one system version at a time, we control its iteration through a system's evolution using our own tool `AsTdEA` ("Architecture Smell and Technical Debt Evolution Analyzer"). `AsTdEA` tracks inter-version smells according to our concept (cf. Section 3) and computes additional properties that require data over multiple versions. The source code and compilation of both the modified `Arcan` and `AsTdEA` (Gnoyke et al., 2023) are available in our replication package.[1]

4.4. Workflow

As depicted in Fig. 13, using our toolchain and the input dataset, we generated an output dataset to answer our research questions, which is also part of our replication package.[1] For each subject system, our output dataset consists of inter-version and intra-version data. The former includes metrics for the entire system, as well as properties of each inter-version smell including references to the intra-version smells that they consist of. Subdivided by releases, the latter contains metrics and all intra-version smells associated to a particular release. Data on an intra-version smell includes its intra-version smell properties (cf. Section 3.4) as well as a reference to the components that are affected.

To extract information and derive conclusions from the dataset, we wrote Python queries that aggregate data, calculate statistical indicators, and generate visualizations. All of these queries are included in our replication package, too. We describe the particular data that we queried for each research question in the beginning of each respective subsection in Section 5.

5. Results and discussion

In this section, we present and discuss our results, based on the research questions we defined in Section 4.1. For this purpose, we outline the data we analyzed to answer each question, describe and summarize the consequent results, as well as discuss their implications.

5.1. $RQ_{1.1}$: Merging and splitting

Data Analysis. To answer $RQ_{1.1}$, how merging and splitting impact the evolution of cyclic dependencies, we queried all intra-version cyclic dependencies alongside their properties, transitions (cf. Section 3.3), and resulting cyclic-dependency families. We investigated class-level and package-level cyclic dependencies separately to identify patterns that only apply to either of these two types. In the following, we provide detailed information from different perspectives based on this data to contribute a comprehensive overview. For clarity, we provide further information on the analyzed data together with the corresponding results.

Results. In Table 3, we summarize the numbers for all cyclic dependencies we identified in our dataset. We observed merges and splits (i.e., a maximum family width > 1) only in 20 (1.3%) of all 1578 class-level inter-version cyclic-dependency families. This number increases to 5822 (23.3%) when considering the class-level intra-version cyclic dependencies that are part of such non-trivial cyclic-dependency families—in total 24,955 occurrences across all families. Consequently, we conclude that cyclic-dependency families with non-trivial evolution graphs tend to have considerably higher family orders compared to cyclic dependencies that do not merge or split. To be precise, the

Table 3
Overview of the cyclic dependencies within our dataset, and those involved in merging and/or splitting.

Scope	Class-level				Package-level			
	Inter-version		Intra-version		Inter-version		Intra-version	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
One-dimensional families	1,558	98.7%	19,133	76.7%	69	90.8%	1,031	70.0%
Families with merges/splits	20	1.3%	5,822	23.3%	7	9.2%	442	30.0%
Total	1,578		24,955		76		1,473	

Table 4
Counts and shares of transition types in cyclic-dependency evolution graphs within our dataset.

type	Class-level		Package-level	
	Absolute	Relative	Absolute	Relative
Pure	22,772	97.2%	1,357	97.1%
Merge	390	1.7%	33	2.4%
Split	281	1.2%	8	0.6%
Both merge & split[a]	19	0.1%	0	0.0%
Total	23,424		1,398	

[a] counted twice; must be subtracted from total.

median family order of the former is 81.5, while it is only seven for the latter. On the package-level, seven (9.2%) of all 76 inter-version cyclic-dependency families merged or split at least once, with 442 (30.0%) of all 1473 intra-version cyclic dependencies belonging to such families. The median family order across non-trivial cyclic-dependencies is 37, whereas it is 10 for one-dimensional cases.

Another perspective on the evolution of cyclic dependencies are transitions in cyclic-dependency evolution graphs, for which we provide an overview in Table 4. We observed a total of 23,424 class-level transitions. Out of these, 22,772 (97.2%) are pure. The remainder represents 390 (1.7%) merge and 281 (1.2%) split transitions, which occurred in 102 merges and 55 splits. So, merges and merge transitions can be observed more frequently than splits and split transitions. Moreover, 19 (0.1%) edges in class-level cyclic-dependency evolution graphs are merge and split transitions at the same time. On the package-level, 1398 transitions occurred, 1357 (97.1%) of which are pure. We observed 33 (2.4%) merge transitions and eight (0.6%) split transitions spanning 15 merges and 4 splits. Consequently, the relative difference between the frequency of merges and splits is considerably larger for package-level than for class-level cyclic dependencies. Not a single package-level transition was both a merge and split transition simultaneously.

In Fig. 14, we illustrate how many class-level intra-version smells participated in merges and splits. Each of the two plots contains a *cumulative distribution function* (CDF) for two perspectives:

- *unweighted*: every merge and split has the same weight, regardless of the number of its transitions.
- *weighted*: every merge and split is weighted by the number of involved transitions, representing the number of intra-version smells that either merge into each other or split from the same intra-version smell.

The data reveals that the majority (≈60%) of both class-level merges and splits involve just two intra-version cyclic dependencies. Moreover, less than 10% of both merges and splits involve more than 10 intra-version cyclic dependencies. We found that the largest merge involves 42 intra-version smells, while the largest split resulted in 101 intra-version smells. Both occurred in the system `Azureus` in the same inter-version smell.

When investigating the weighted CDFs, only about a third of all merge transitions took place in merges with two intra-version cyclic dependencies, while we can see an approximately gradual increase for merges up to approximately 10 incoming edges. Less than a quarter of all split transitions occurred in splits with only two outgoing edges. The outlier in `Azureus` represents approximately every tenth merge transition and about one third of all split transitions. It is also the widest cyclic dependency family in our dataset with a maximum family width of 109 co-existing intra-version smells, as well as the cyclic dependency family with the highest family order of 2500 intra-version smells. This constitutes nearly a third of all class-level intra-version cyclic dependencies in `Azureus`. Meanwhile, on the package-level, every split constituted exactly two outgoing edges, while 80% of the merges (≈73% of merge transitions) involved only two incoming edges (i.e., two intra-version cyclic depedencies merging with each other). The remaining merges did not involve more than three incoming edges.

One way of assessing patterns in cyclic-dependency evolution graphs is to see how family widths develop over time. Therefore, we provide two different perspectives on the evolution of family widths in non-trivial cyclic-dependency evolution graphs in Fig. 15—both for the class-level and package-level:

- *exclusive*: we consider only versions with a family width > 1
- *inclusive*: we consider all versions

In each plot, we show how the median family width, as well as its range, evolve over time; specifically the number of versions since the introduction of the first family member. For this purpose, we aggregate data from all cyclic-dependency families—weighting them equally.

The data on family widths reveals several insights. First, the median family width tends to increase for class-level cyclic dependencies, especially in the exclusive data, while, in comparison, it stagnates for package-level cyclic dependencies. In other words: class-level cyclic-dependency families tend to grow wider over time. Second, extreme family widths tend to occur regardless of the age of the respective smell. Finally, older class-level cyclic-dependency families either have a width of a single branch or widths greater than two branches. We can attribute the maximum class-level family width of 109 to the aforementioned cyclic-dependency family in `Azureus`. The next widest families occurred in `Azureus` with a maximum family width of 32, in `Hibernate` with 22, and in `ArgoUML` with 21.

A pattern that we observed in many non-trivial cyclic-dependency evolution graphs is that a particular path through the graph "attracts" many merges with its intra-version smells—leading to disproportional growth in complexity (i.e., order, size, and number of subcycles). We display an example for this trend in Fig. 16, which contains the evolution graph of a class-level cyclic dependency in the system `Lucene`. It starts with a set of relatively small intra-version cyclic dependencies, lasts for the entire observed period, and ends in a considerably larger cyclic dependency affecting 189 classes (among two small intra-version cyclic dependencies). The graph is also a fitting example for merges involving more than two intra-version cyclic dependencies, as well as

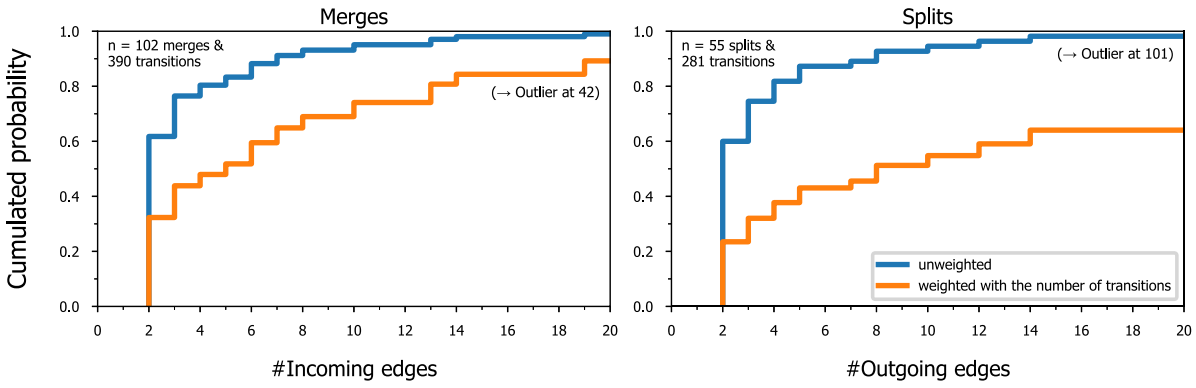

Fig. 14. Distribution of the number of incoming and outgoing edges in merges and splits of class-level cyclic dependencies.

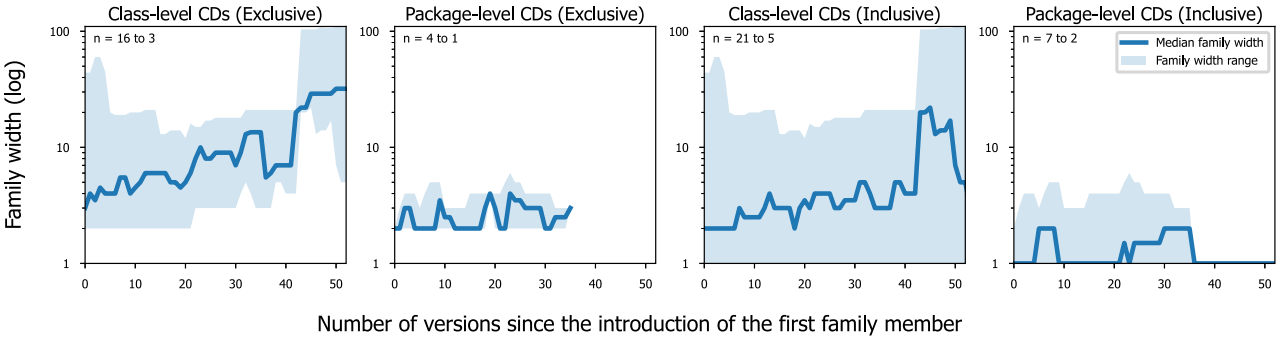

Fig. 15. Evolution of the family width median and family width range in non-trivial cyclic-dependency (CD) evolution graphs.

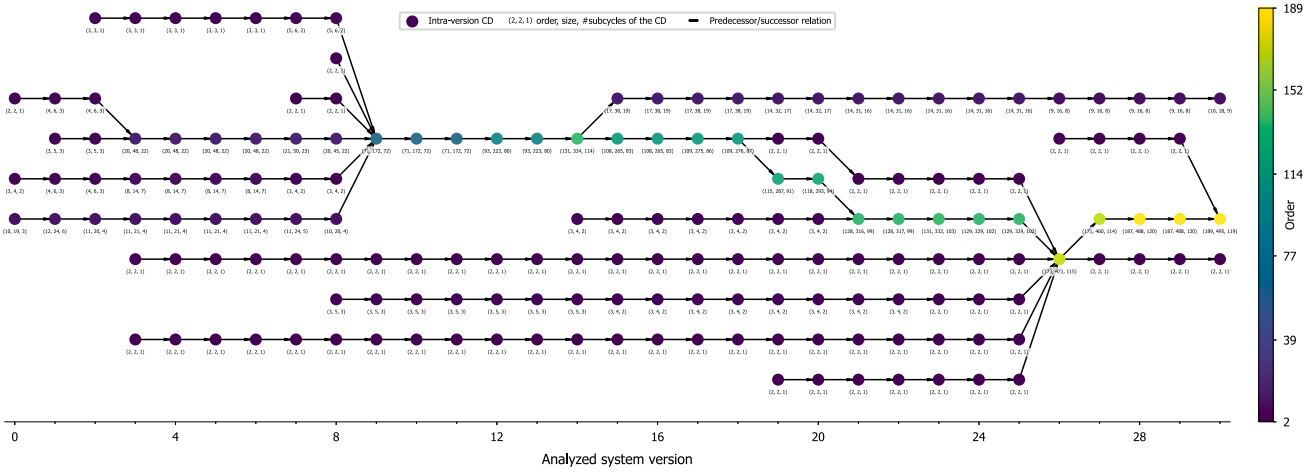

Fig. 16. Example of a class-level cyclic-dependency (CD) evolution graph with merges, splits, and merges of previously split branches in the system `Lucene`, resulting in a large intra-version cyclic dependency and two smaller ones.

branches re-merging after having split. In particular, the latter implies that cyclic-dependency evolution graphs cannot be generalized as trees.

To quantify our observation about increasing inequalities in cyclic-dependency evolution graphs over time, we considered how the order, size, and number of subcycle properties are distributed in co-existing intra-version smells over time. In our example, the order distribution in the last version in Fig. 16 is $\{10, 189, 2\}$. For each of these distributions, we calculated the Gini index to assess how balanced or imbalanced the values are. We considered only versions in cyclic-dependency families with a family width of at least two to enable the Gini index calculation. This corresponds to the exclusive perspective in Fig. 15. We show how the mean Gini value of all non-trivial cyclic dependency families evolves over time for both the order and size properties on both the class-level and package-level in Fig. 17. The light outline represents the confidence interval, and we note the respective alpha value in each plot.

All three properties (we do not display the number of subcycles, but it differs only marginally) follow similar patterns: the results indicate that the Gini index increases over time and eventually plateaus, resulting in approximately a doubling for class-level cyclic-dependency families. This implies that cyclic-dependency families tend to become more imbalanced in their order, size, and number of subcycles the older they become, as we can see in Fig. 16. The graph for package-level cyclic dependencies is shown a lower alpha value and is cut off after a lower number of versions because of a lack of data points, but follows a similar trend in the observed period.

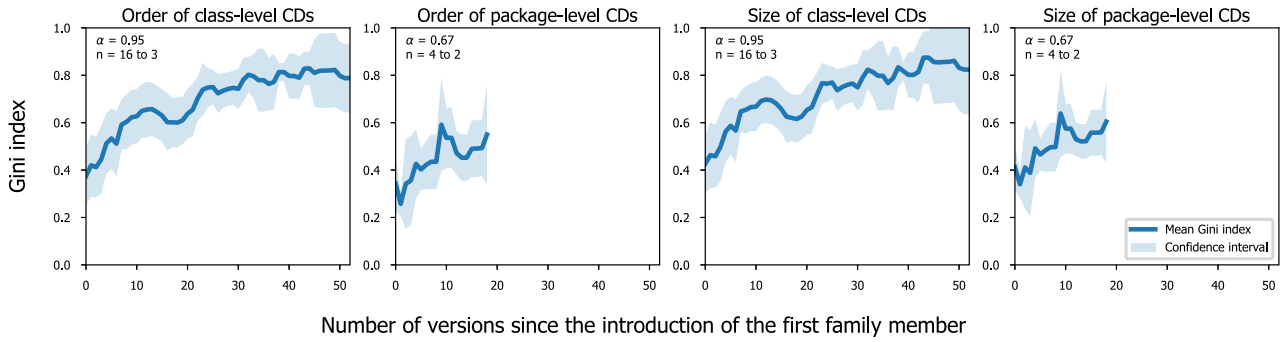

Fig. 17. Gini index evolution for the order and size distributions of intra-version cyclic dependencies (CDs) in the same version of non-trivial cyclic-dependency evolution graphs.

$RQ_{1.1}$: How do merging and splitting impact the evolution of cyclic dependencies?

• Merges and splits occur in few cyclic-dependency families, which tend to encompass many intra-version cyclic dependencies. • Most transitions in cyclic-dependency evolution graphs are pure. • Merges occur more often than splits. • Most merges and splits involve only two intra-version cyclic dependencies, but many merge and split transitions can be seen in larger merges and splits. • The family width of class-level cyclic dependencies tends to increase over time, while it tends to stagnate for package-level cyclic dependencies. • Cyclic-dependency families tend to become imbalanced over time with a single path accumulating many merged cyclic dependencies, resulting in high numbers for order, size, and subcycles. • Merges, on average, result in an increase in the number of subcycles, while splits do not exhibit a consistent trend. •

Table 5
Average relative change in the number of subcycles in cyclic dependencies during merging and splitting ($\alpha = 0.95$).

	Class-level	Package-level
Merging	+40% ± 20%	+38% ± 22%
Splitting	+0.7% ± 8.2%	−34% ± 89%

To assess how merges and splits affect the overall complexity of a system, we analyzed whether they significantly affect the total number of subcycles. For this purpose, we compared the set of intra-version smells before a merge with the intra-version smell after the merge and vice versa for splits. As an example, the split in version 14 to 15 in Fig. 16 reduces the number of subcycles from 114 to 102, as it splits into two intra-version smells with 83 and 19 subcycles. To avoid potential biases caused by large smells, we computed the relative difference in subcycles, which is approximately −11% in this example. When considering all merges and all splits in all systems of our dataset with an alpha value of 0.95, we obtain the results in Table 5. We can see that merges cause, on average, a considerable increase in the number of subcycles, while we did not find a consistent trend for splits, regardless whether class-level or package-level.

Discussion. Our results indicate that merges and splits in cyclic dependencies are a comparatively rare event, since only few cyclic-dependency families involve the respective transitions. The low number of merge and split transitions can be explained by the observation that most branches in cyclic-dependency evolution graphs persist for several versions without merging or splitting. Thus, merge and split transitions are a subset of non-trivial cyclic-dependency evolution graphs that are a subset of all cyclic-dependency families. Still, ignoring merges and splits because of these observations would result in a considerably skewed picture considering the high family orders of non-trivial cyclic-dependency families. In other words:

> *About a quarter of intra-version cyclic dependencies on the class-level and a third on the package-level would be tracked incorrectly if merging and splitting were not taken into account.*

The distribution of incoming and outgoing edges in merges and splits follows a similar pattern. While only a minority of merges and splits involve more than two intra-version smells, most merge and split transitions occur within larger merges and splits. We argue that:

> *Developers should be aware of large-scale merge events, as they can considerably increase the cyclic dependencies' and ultimately the system's complexity.*

At first glance, our observation that class-level cyclic-dependency families tend to grow wider over time seems to contradict the fact that we observed more merges than splits. Using Fig. 16 as an example, we can see a possible explanation: In the depicted cyclic-dependency evolution graph, not a single branch ends because the last intra-version smell was removed from the system. Instead, new branches are introduced in various versions due to the manifestation of a new cyclic dependency without predecessors. When in each version, on average, more branches are created this way than merged into other branches, the family width increases over time.

> *We thus suggest that developers should not ignore newly introduced cyclic dependencies for too long, as they otherwise may merge into large cyclic dependencies, rendering refactoring harder and harder.*

We can draw a similar conclusion for the trend in cyclic-dependency families of developing a main path with considerably higher orders, sizes, and numbers of subcycles.

> *If large intra-version cyclic dependencies are not actively reduced in complexity, they tend to attract merges, and thus grow even larger over time.*

Tool support that regularly checks for cyclic-dependency merges and growing instances could warn developers in advance before wide spanning and tangled smells propagate themselves through the system.

We consider few large cyclic dependencies as more critical than many small cyclic dependencies that affect the same set of components. Larger cyclic dependencies mean that changes propagate through many more parts of the system with more tangled and complex inter-dependencies between components. Furthermore, resolving small cyclic dependencies requires removing only a small number of dependencies—if not a single one, which follows the *divide and conquer* principle. We can derive empirical evidence for this from the significant

increase in the number of subcycles in merges: Each subcycle represents a possible path for change propagation, causing ripple effects. A merge can easily cause many new subcycles in an unchanged set of affected components, and thus decreases the components' maintainability. Interestingly, splits do not seem to reduce complexity in the same way that merges increase it, given their insignificant impact on the number of subcycles. Also considering that merges occur more often than splits, we stress that:

> *Merges and splits do not cancel each other out, which is why developer intervention is needed to refactor cyclic dependencies and stop their growth.*

5.2. $RQ_{1.2}$: Cyclic-dependency shapes

Data Analysis. To answer $RQ_{1.2}$, how the shape of cyclic dependencies evolves over time, we investigated all transitions between intra-version cyclic dependencies. For every transition, we queried the following data:

- the type of transition: pure, merge, or split;
- the shape of the head vertex (i.e., older intra-version cyclic dependency); and
- the shape of the tail vertex (i.e., younger intra-version cyclic dependency).

Using this data and depending on the type of transition, as well as the level of cyclic dependency (i.e., class or package), we determined

- how many transitions occurred from which shape to which shape,
- how likely a particular shape transitions into a particular shape, and
- how likely a particular shape participates in merges and splits, both incoming and outgoing.

So, we could identify to what extent shapes transition into each other, or remain stable throughout the life-cycle of a cyclic dependency.

Results. In Table 6, we summarize the primary results for our analysis of the shape evolution in cyclic dependencies. Specifically, Table 6 represents a *1st-order right stochastic matrix* per combination of level (i.e., class-level and package-level) and transition type (i.e., pure, merge, and split). This means that every percentage represents the observed probability of a particular shape (row) transitioning into a particular shape (column) in the following version of a system. Every row adds up to 100% (rounding errors may lead to slight deviations). So, Table 6 represents the tabular equivalent to a *markov chain* (Gagniuc, 2017). We do not show semi-cliques in Table 6, because we classified only a single intra-version package-level cyclic dependency as such. It transitioned into a multi-hub in the following version. When referring to "all shapes" in the following, we therefore do not include semi-cliques.

A large majority of pure transitions did not lead to a change in the shape of cyclic dependencies. For example, 99% of all class-level tiny intra-version cyclic dependencies remained tiny in the following version. On the package-level, this ratio of cyclic dependencies remaining in the same shape after pure transitions was slightly lower, with down to 89% for unknown shapes. If pure package-level transitions did not result in the same shape, they most often led to a multi-hub, which was not the case for class-level cyclic dependencies.

When looking at merge transitions, we can see that all package-level and most (89% or more) class-level transitions lead to multi-hubs. Merges that did not lead to multi-hubs led to star shapes and two unknowns (from tiny cyclic dependencies). Class-level multi-hubs that merged with other cyclic dependencies remained a multi-hub in 99% of cases. Interestingly, all shapes on the class-level and almost all shapes on the package-level (except cliques and chains) merged at least once with an intra-version cyclic dependency of another shape.

The stochastic matrices of split transitions indicate almost the opposite compared to merge transitions. Here, only stars, multi-hubs, and unknowns on the class-level, as well as multi-hubs on the package-level split into multiple cyclic dependencies in the following version. On the class-level, 97% of split transitions occurred from a multi-hub. Meanwhile, splits resulted in a diverse set of shapes. Around a third of the class-level and half of the package-level split transitions from multi-hubs resulted in another multi-hub.

To evaluate the likelihood of a specific shape participating in a merge and/or split, we show their share of merge and split transitions in Fig. 18. On the class-level, the most likely shape to participate in merges are cliques, as one eighth of the transitions we observed from cliques are merge transitions. Since very few cyclic dependencies were classified as cliques and we only registered a single merge from a clique, this entry should be considered with caution. Among the remaining shapes on the class-level, by far the most likely shape to participate in merges are multi-hubs with ≈4% of transitions. On the package-level, the most likely shape to merge are unknowns with ≈5% of merges stemming from unknown shapes, followed by circles and multi-hubs. Out of these, only merges from multi-hubs occurred more than once. At the same time, the only shape with a considerable share of merge transitions among all incoming transitions, both on the class- and package-level, are multi-hubs. For instance, more than every tenth class-level transition into multi-hubs is a merge transition.

About 8% of transitions from class-level multi-hubs are split transitions, and thus twice as many as merge transitions from multi-hubs. With 1.5%, transitions from package-level multi-hubs are considerably less likely to be split transitions. On the class-level, multi-hubs are the shape with the highest share of split transitions among all of their incoming transitions (2.5%), followed by stars. In contrast, on the package-level, 5.6% of transitions to unknowns are split transitions, followed by circles (both only one data point). Overall, disregarding the singular clique-merge transition, the shape with the lowest shares of pure incoming and outgoing transitions on the class-level is the multi-hub with ≈88 and 87%. For the package-level, multi-hubs and unknowns have comparably low shares (i.e., ≈95 to 93%) of pure transitions.

In total, 92 class-level merges (≈90% of all class-level merges) resulted in a multi-hub. Out of those, in ≈92% (85) of the cases, at least one multi-hub was already present before the merge. In other words, most merges caused an existing multi-hub to grow. On the package-level, every one of the 15 merges we observed resulted in a multi-hub, and in all cases a multi-hub was already present. This means that at least one architecture smell that participated in the merge transition already had the shape of a multi-hub.

We can perform a similar analysis for splits. On the class-level, 51 splits (≈93%) of all class-level splits occurred from a multi-hub. Out of those, all except one split resulted in at least one multi-hub after the split. Thus, most splits caused an existing multi-hub to shrink. As we can see in Table 6, all four package-level splits emerged from a multi-hub. In three cases, a multi-hub remained after the split.

Discussion. The low shape-change rate during pure transitions we observed aligns to our intuition: Usually, a considerable change in the set of components affected by a cyclic dependency or the dependencies among these components would be required to change the shape of the smell. If a change leads to the expansion of a cyclic dependency with additional components, a merge occurs if one of these components is already affected by a cyclic dependency and no dependencies are removed. In contrast, removing affected components or dependencies between components in a cyclic dependency potentially leads to a split. It requires more detailed (and qualitative) analyses to determine the reason why class-level cyclic dependencies maintain their shape during pure transitions more often than package-level cyclic dependencies;

Table 6
The 1st-order right stochastic matrices for shape transitions of related class-level and package-level cyclic dependencies in adjacent versions in our dataset. Note that we observed empty transitions zero times, whereas transitions with 0% occurred at least once.

	from ↓	class-level cyclic dependencies								package-level cyclic dependencies							
	to →	tiny	clique	circle	chain	star	m.-hub	unkn.	Total	tiny	clique	circle	chain	star	m.-hub	unkn.	Total
Pure transitions	tiny	99%		1%	0%	0%	0%	0%	10,296	98%	0%	1%	0%		1%	0%	571
	clique		100%						7		99%		1%				82
	circle	1%		98%	0%	1%	0%	0%	3,734			95%		3%	2%		60
	chain	0%		1%	97%	0%	1%		404				93%		7%		29
	star	0%		0%	0%	99%	0%		5,144			1%	1%	92%	5%		78
	m.-hub		0%	0%	0%	0%	99%	0%	2,913				0%	1%	99%		518
	unkn.			2%			1%	97%	274						11%	89%	18
Merge transitions	tiny					7%	91%	2%	114						100%		12
	clique						100%		1								0
	circle					11%	89%		46						100%		2
	chain					11%	89%		9								0
	star					9%	91%		87						100%		1
	m.-hub					1%	99%		129						100%		17
	unkn.						100%		4						100%		1
Split transitions	tiny								0								0
	clique								0								0
	circle								0								0
	chain								0								0
	star	33%		17%		50%			6								0
	m.-hub	23%		10%	0%	35%	32%	0%	273	25%		13%			50%	13%	8
	unkn.	50%		50%					2								0

(m.-hub = multi-hub, unkn. = unknown, total = number of observed edges in cyclic-dependency evolution graphs).

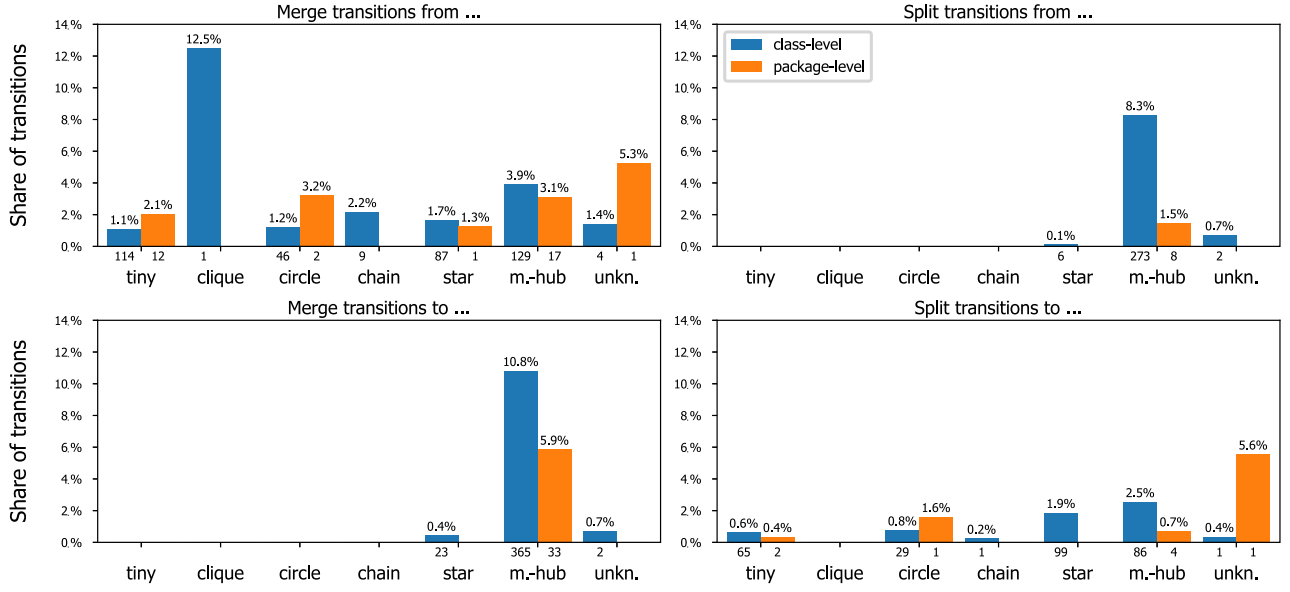

Fig. 18. Share of outgoing merge and split transitions from cyclic-dependency shapes in relation to their total number of outgoing transitions, as well as the share of incoming merge and split transitions to cyclic dependency shapes in relation to their total number of incoming transitions. We display the absolute number of observed transitions of each type below the bars. Note that the provided percentages and absolute numbers refer to the share and number of specific transition types from/to each shape individually. For instance, the share of 1.1% merge transitions and 0% split transitions from tiny instances implies that 98.9% transitions outgoing from tiny instances were pure. In absolute numbers: 10,296, as given in Table 6.

$RQ_{1.2}$: How does the shape of cyclic dependencies change over time?

• The shape of cyclic dependencies remains mostly the same during pure transitions. • Most merge transitions result in multi-hubs and specifically in the growth of existing multi-hubs involving merges with various other shapes. • Splits usually originate from multi-hubs and result in a smaller multi-hub plus one or more intra-version cyclic dependencies of various other shapes. • Transitions from and to multi-hubs are particularly likely to be merge or split transitions compared to other shapes. •

and why only package-level cyclic dependencies tend to change into multi-hubs during pure transitions. Potential reasons may be different patterns in introducing dependencies among packages than among classes during software development.

Merge transitions to tiny cyclic dependencies or split transitions from them are not possible given that a single component cannot constitute a cyclic dependency, which is why we did not observe such transitions. Similarly, we did not find merges into a circle or a circle splitting, which we consider unlikely based on intuition—since a very specific introduction or removal of dependencies is required during the transition. The lack of merge and split transitions from and to cliques can be explained with their low frequency in our dataset.

It appears logical that merges often lead to multi-hubs, given their polycentric and complex shape. Following this reasoning, their relatively frequent splitting is also comprehensible: Multi-hubs consist of multiple clusters of components that are only interconnected by a low number of components and dependencies. As such complex structures are probably not intentionally designed by developers, refactorings can lead to rearranging the easily breakable dependencies between the clusters of components. The fact that chains and unknowns almost never were the result of merges or the origin of splits seems counter-intuitive and requires further investigation, given that, for example, the result of a merge could be complex enough to be classified as an unknown. Stars almost never being split could be explained by them being introduced intentionally, as suspected in a related case study on architecture smell evolution in C++ systems by Sas et al. (2022b).

We regard the finding that merges and splits often constitute the growth and shrinking of an existing multi-hub that continues to exist as important. It means that the already complex multi-hub shape tends to spread to more components, involving an increasing number of subcycles, and adding even more clusters of components to the supercycle. This underpins our conclusions in Section 5.1 that:

> *Developers should be aware of cyclic-dependency merges to avoid that the system quality degrades.*

At the same time, as multi-hubs are rarely resolved by splits, cyclic-dependency splitting often does not result in reduced complexity, even though it does represent the divide and conquer paradigm.

The study by Sas et al. can also serve as a frame of reference for our findings, as the authors analyzed how cyclic-dependency shapes transition into each other as well. Still, there are major differences in the experimental setup, specifically the dataset (open-source vs. industrial), the language of the subject systems (Java vs. C++), the definition of cyclic-dependency instances (supercycle vs. subcycle), the consideration of cyclic-dependency merges and splits by us, and the set of detected cyclic-dependency shapes (multi-hubs, semi-cliques, and unknowns were not included by Sas et al.). In our study, ≈96% of class-level and ≈94% of package-level transitions were pure and did not result in a change of shape. Sas et al. report only a 73% rate of transitions without shape changes. They found stars and chains to be especially prone to changes, whereas circles showed the lowest relative change rate, which is not the case in our results. A possible explanation for these divergent results is the set of listed differences in the experimental setup. This is supported by the share of shapes among the entire cyclic-dependency population reported by Sas et al., which highly differs from our results. For example, they found 86% circles in their dataset, while for us, tiny cyclic dependencies were the most frequent shape with 45% of intra-version smells on the class-level and 42% on the package-level.

5.3. RQ_2: *Architecture-smell properties*

Data Analysis. To answer RQ_2, how the properties of architecture smells evolve over time, we examined the distribution of each property's values depending on the age of the corresponding smells. For this purpose, we used a set of typical quantile values (0.25, 0.50, 0.75, 0.90, 0.95, and 0.98) for every numerical property and every smell age depending on the type of the architecture smell. These values include the quartile values, which provide an overview of the general trend, as well as samples for extreme values. The latter enable us to assess whether extreme values are more prevalent for older smells. Extreme values are of interest because they usually have a higher relevance for the technical debt in a system and require more refactoring effort. While this quantile analysis does not provide information on the evolution of individual smells and their properties, it provides an overview about the distribution of a property's values depending on the age of smells. We analyzed our data through four different perspectives:

- $\mathbf{P_1}$ smell age assuming that a smell existent in the first version we cover was introduced in that version.
- $\mathbf{P_2}$ smell age without considering inter-version smells that are present in the first version covered in our data.
- $\mathbf{P_3}$ remaining smell age assuming that every smell still existent in the last version covered in our data was removed in that version.
- $\mathbf{P_4}$ remaining smell age without considering inter-version smells that still exist in the last version in our data.

Specifically, we used perspectives $\mathbf{P_2}$ and $\mathbf{P_4}$ to account for the assumption that the results may be rendered incorrect by including smells whose actual age and remaining age are unknown. We used perspectives $\mathbf{P_1}$ and $\mathbf{P_3}$ to account for all architecture smells in our dataset and to provide a reasonable estimate about each smell's evolution until its (potential) removal. During our analysis, we found that the results for all four perspectives are highly similar across most properties, which is why we focus on presenting and discussing them based on $\mathbf{P_1}$ in the following. The details of all perspectives are part of our replication package.[1]

Results. In Fig. 19, we provide an overview of how the different properties of **class-level cyclic dependencies** evolve throughout the versions in our dataset. We can see that for all measures of complexity (i.e., order, size, severity score, technical debt, number of subcycles), the range of values in our dataset increases with a higher smell age. The lower bound usually remains stable, for instance, the 0.25 quantile for the order or size, while the upper quantiles become more extreme over time. Especially the 0.98 quantile increases substantially for the measures of complexity, usually by an order of magnitude within the time span we consider. For instance, it increases from 23 to 168 for the order, from 46 to 456 for the size, and from 21 to 113 for the number of subcycles. Considering the technical debt, only the extreme quantiles increase, while lower quantiles even reduce over time. Meanwhile, the centrality increases slightly over time, the overlap ratio remains zero for most instances, and the relative number of inheritance edges remains mostly stable. The absolute number of inheritance edges shows a similar trend to the size property. Lastly, the share of tiny class-level cyclic dependencies decreases with an increasing smell age, while the shares of multi-hubs and stars grows. The number of intra-version smells that we examined for these results decreases from 1819 with an age of zero versions to 253 with an age of 27 versions.

We provide equivalent overviews to Fig. 19 for all other types of architecture smells and perspectives in our replication package. For **package-level cyclic dependencies**, we can generally identify similar patterns to class-level cyclic dependencies. The 0.90, 0.95, and 0.98 quantiles increase over time for the measures of complexity. For the 0.98 quantile from a smell age of zero to 27 versions, this includes the order rising from 35 to 232, the size from 255 to 1919, and the number of subcycles from 130 to 1057. These numbers are all higher than for class-level cyclic dependencies. At the same time, the number of datapoints (i.e., intra-version smells with particular ages) is lower, with 93 to 20 in the covered time period. For several properties of package-level cyclic dependencies, the median increases at first with an increasing age, while dropping again around an age of 20 versions. Especially the size overcomplexity is interesting, because the median newly created instance does not have any excess edges, while this number rises to more than 50% at an age of nine versions. Unlike for class-level cyclic dependencies, we identified a decrease in the median of several properties for higher ages. Moreover, the rise in centrality at higher quantiles, as well as the increase in the share of multi-hubs and the decline in the share of tiny shapes, are more pronounced for package-level cyclic dependencies compared to the class-level.

The evolution of **hub-like dependency** property quantiles differs considerably from cyclic dependencies. Most quantiles grow slowly over time or stagnate, while the relative difference of low to high quantiles did not increase for most properties. In the measures of complexity (e.g., the order, size, or severity score), extreme quantiles like 0.98 remain mostly stable, while lower quantiles like 0.90 and 0.75 grow with an increasing smell age. For example, the order's 0.75

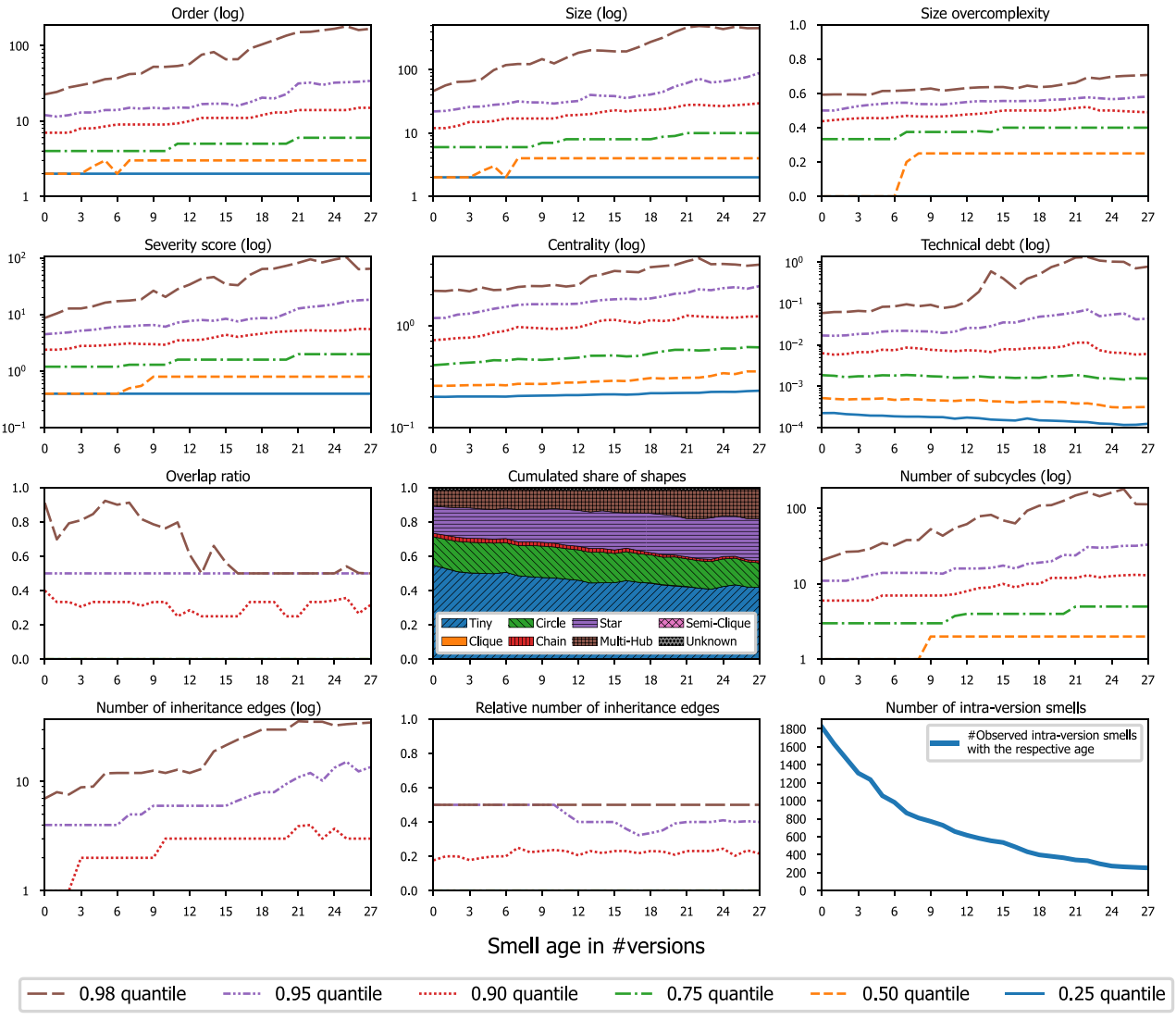

Fig. 19. Quantile evolution of class-level cyclic dependency property values.

quantile increases from 81 to 156 in the period we cover, while it increases from 472 to 1287 for the size. Moreover, hub-like dependencies are the architecture smell type with the least diverse distribution of size overcomplexities, whose quantiles remain mostly stable and close to each other. For instance, the two central quartiles remain entirely within the bounds of 0.75 to 0.9. The centrality of hub-like dependencies shows a slowly increasing trend, while the overlap ratio remains stable with a decreasing range between major quantiles. Lastly, the median hub ratio slightly grows over time, which means that hub classes slowly shift towards more incoming than outgoing edges. The number of hub-like dependencies per age that we considered for this analysis shifts from 158 to 14.

Most property quantiles for **unstable dependencies** slowly increase with an increasing smell age—and for most properties the key quantiles show similar trends. For example, the 0.98 quantile of the order increases from 8 to 20 in the period we cover, while it increases from 19 to 59 for the size. Median unstable dependencies in our data never feature excess edges, and thus their size overcomplexity equals zero, while higher quantiles show increasing to fluctuating trends. The centrality slightly increases in all quantiles over time, while most quantiles of the technical debt stagnate or decrease. For any smell age, most unstable dependencies overlap entirely with other unstable dependencies or package-level cyclic dependencies. Moreover, the DoUD and the instability-gap median both increase over time in most quantiles. The range of the IQR also increases along the smell age. Eventually, the number of considered intra-version unstable dependencies decreases from 885 with an age of zero to 246 with an age of 27 versions.

Discussion. By aggregating all smell instances with the same age to calculate quantiles, we trade details of individual smells for a more comprehensive overall understanding of the evolution of architecture-smell properties. This perspective introduces a possible bias: with an increasing smell age, the dataset contains fewer instances. For example, if small smells tend to be removed faster, it may appear that smells grow over time, even if bigger instances remain unchanged in size or order. In our previous work (Gnoyke et al., 2021), we observed increasingly complex class-level cyclic dependencies to persist longer. Similarly, Sas et al. (2019, 2022b) found through two studies that around 30 to 50% of cyclic, hub-like, and unstable dependencies remain rather constant regarding order and size during their lifespan. However, a considerably higher share of the remaining instances grew over time as opposed to reduce in order and size. These findings indicate that our aggregation is feasible, and thus we argue that we can draw conclusions from the general trends in our results. Moreover, insights on the distribution of property values depending on the smell age are a novel and helpful result on their own. The fact that all four perspectives we outlined yielded similar trends further supports the validity of our

RQ_2: How do the properties of architecture smells evolve over time?

• For cyclic dependencies, extreme values in properties of complexity, such as the order, size, number of subcycles, severity score, or technical debt, become more prevalent with an increasing smell age. • Hub-like dependencies are the architecture smell type with the least change over time regarding their properties. • Most properties of unstable dependencies grow slowly over time in most quantiles. • For all architecture smell types, the centrality tends to grow over time. •

results. Given that there is little difference between querying by smell age since introduction and the remaining smell before removal, we conclude that:

> *Architecture-smell instances share common trends as they grow older.*

An impending removal seems to be less pivotal on the values of architecture-smell properties.

We argue that the increasingly extreme upper quantile values that arise over time for the measures of complexity of cyclic dependencies are an indication that:

> *An early identification and targeted removal of smell instances is critical to keep a system maintainable.*

If a large cyclic dependency is not detected or left untreated, it tends to merge with other instances as we demonstrate in Section 5.1. This results in a complex tangling with many affected components, edges between them, and subcycles propagating changes to components. The growing share of multi-hubs and stars supports our results in Section 5.2, highlighting increasing complexity in aging smells.

The increasing hub ratio of hub-like dependencies implies that:

> *Over time, more and more classes reference the hub class, which undermines the principle of modularity.*

A possible explanation is that it may seem easier to developers to fetch information directly from the hub class instead of introducing a specialized interface when a new feature is being introduced.

While the absolute quantile values of some unstable dependency properties are considerably less extreme than for cyclic dependencies, their increasing trends highlight that:

> *Also unstable dependencies require a constant monitoring and targeted refactoring.*

Otherwise, the change propagation needed becomes stronger over time—especially given that the measures of instability (degree of unstable dependency and instability gap) also tend to increase with the smell age.

Our conclusions are further supported by the increasing trend in the centrality of all types of architecture smells. This is caused by a smell being located more centrally in a system, and thus ripple effects becoming more likely. Analyzing the evolution of architecture-smell properties opens several new questions for future work. This includes finding reasons for specific trends, such as the increasing centrality over time or why the instability measures in unstable dependencies tend to become more pronounced.

5.4. RQ_3: *Compound interest*

Data Analysis. To answer RQ_3, how architecture smells and technical debt relate to compound interest, we performed a regression and a correlation analysis. For both, we defined the following six key variables that we tracked through the evolution of each subject system:

1. the total number of architecture smells,
2. the technical debt index (TDI),
3. the number of components impacted by all smells,
4. the total order (i.e., the sum of all smells' orders),
5. the total size (i.e., the sum of all smells' sizes), and
6. the total number of subcycles (i.e., the sum of all cyclic dependencies' subcycles).

In every version, we determined the absolute and a normalized value for each of these variables. The latter serves as a means for us to investigate how relative levels of architectural problems evolve over time. This can be helpful to eliminate a potential bias caused by system growth, which is likely to cause key variables to increase. Given that each of the architecture-smell types in our study is defined as a set of components interacting in a specific way, we normalize values by dividing the absolute value with the number of components in a version (i.e., sum of classes or packages).

We chose the above six variables, because they provide insights on architectural erosion from different points of view. In detail, we expect that the effort to refactor increases with the number of individual issues (in this case architecture smells), but also with the overall severity of the issues, represented by the technical debt index. The more components are affected by smells, the easier changes may ripple through a system. If a component is affected by multiple smells, we expect this effect to be more distinct, which is represented by the total order. We can furthermore assume that more connections between components accelerate the propagation of changes, which the total size accounts for. Eventually, as we discuss in Section 5.1, each subcycle constitutes a possible path for ripple effects, which renders their total number an interesting subject for our analysis.

Regression Analysis. If the metaphor of compound interest applies to architecture smells and the consequent technical debt, we can assume the key variables to grow exponentially over time. We tested whether this is the case using a regression analysis, in which we considered the progression of time (i.e., the version number) of each system as the independent variable and the key variables as the dependent variables. To obtain a more comprehensive overview, we performed the regression analysis both for the absolute and normalized variables. Consequently, we analyzed $14 \cdot 6 \cdot 2 = 168$ (systems $\cdot$ variables $\cdot$ absolute/normalized) data series.

For every combination, our null hypothesis is that the dependent variable does not grow exponentially. To evaluate the hypothesis, we applied a logarithmic transformation to each dependent variable and performed a *simple linear regression* to obtain the regression line's gradient. Additionally, we extracted the slope's p-value and *coefficient of determination* i.e., R^2. We selected a minimum R^2 value of 0.5 as the first criterion to reject the null hypothesis, as only in this case the majority of the data's variance can be explained by exponential growth (Fahrmeir et al., 2013). Furthermore, only positive slopes mean that the dependent variable is increasing, marking the second criterion. Additionally, we consider only significant slopes. Given the high number of hypotheses, a multiple test correction is necessary to control the number of random null hypothesis rejections. For this purpose, we employed the *Benjamini–Hochberg method* (Benjamini and Hochberg, 1995) and set the *false discovery rate* (FDR) to 5%. We deemed this false discovery rate an acceptable compromise of false-positives and false-negatives, given the various criteria to accept the null hypothesis in this analysis. We chose this method, as it reduces the number of false-negatives in comparison to other multiple test correction methods.

Afterwards, we performed a manual validation on the remaining data series to check whether the actual graphs resemble exponential growth. For this purpose, we compared the expected values assuming exponential growth with the actual observations. Cases in which the data grows over time, but follows a more logarithmic line constitute

linear or polynomial growth because of the initial logarithmic transformation. Only when we could not observe such patterns, we rejected the respective null hypothesis, and thus assume exponential growth.

Correlation Analysis. While we tested for the existence of exponential growth in the regression analysis, we evaluated the existence of a more general relationship between the severity of existing issues in the code and the aggravation of issues — be it old or new ones — using a correlation analysis. This is based on the hypothesis that a high level of technical debt, as well as high architecture smell quantity and severity, amplifies the manifestation of new and growth of existing technical debt and architecture smells. For this analysis, we considered only the normalized variables, as large systems can be expected to both have a higher absolute number of issues, as well as a quicker absolute rate of introducing new issues. Therefore, absolute variables would introduce a bias towards seemingly high correlations.

Similarly to the regression analysis, we tested for various combinations of variables to obtain a comprehensive understanding of the data. Specifically, in each combination, the independent variable was one of the six key variables we defined above. The normalized growth of one of the six key variables served as the dependent variable (i.e., relating each variable to every other variable and itself). To determine a variable's growth, we considered the following with respect to each system version:

- A newly introduced smell → Take the value of the key variable.
- A smell whose key variable grew → Take the delta of the key variable.

In the case of cyclic-dependency families, we summed up the key variable over all parallel branches in the same version to determine the growth. We did not consider smells that have been removed or that reduced in size in the correlation analysis to account for the effect of deliberate refactoring, which would reduce the number and severity of issues.

The simplest method to discover causal effects would be to compare the independent variable's value with the dependent variable's value in the subsequent version. We realized that doing so would induce a considerable degree of noise, and thus could result in very few significant correlations, since many of our subject systems fluctuated locally within otherwise clear trends. For smoothing out local fluctuations and eliminating their effect on the global trend, we employed a *low-pass filter* to both the independent and dependent variable. Specifically, we employed a *moving average* with *binomial weighting*, which weights the central data point of the sampling width the highest (Nixon and Aguado, 2002). While this causes a certain overlap between two adjacent data points in the correlation analysis, we regard this as an acceptable trade-off to investigate causalities. We found a sampling width of seven versions to have the best effect on suppressing local fluctuations and limiting information loss.

As different systems can be expected to exhibit varying levels of technical debt as well as architecture smell numbers and severity, we tested each system individually. Ultimately, we tested $14 \cdot 6 \cdot 6 = 504$ (systems · variables · variables) combinations. Our null hypothesis for every data series is that there is no positive correlation between the independent and dependent variable. Similarly to the regression analysis, we performed a Benjamini–Hochberg multiple test correction based on the correlations's p-value to account for the high number of hypothesis tests. We again set the false discovery rate to 5% to avoid that we miss correlations, given that we are mainly interested in the aggregated picture and less in individual correlations. As we do not test for linear correlation, but the presence of a correlation in general, we decided to use a *rank correlation* procedure. Specifically, we employed the rank correlation by Spearman (1904), because of its wide use in research (Linebach et al., 2014) and related studies (Sas et al., 2019).

Results. Next, we describe the results of our regression and correlation analysis.

Regression Analysis. Out of the 168 data series for the regression analysis, we observed an R^2 of least 0.5 in 104 cases. Of these, 85 regression lines showed a positive gradient, which all had a lower p-value than the Benjamini–Hochberg corrected threshold. Even when lowering the false discovery rate to, for example, 1%, this was still the case. By examining the data, we can see that the null hypothesis was only accepted according to the multiple hypothesis test in data series with a negative slope or with low R^2 values. We eventually accepted exponential growth in 53 cases after our manual validation, which involves 43 absolute and 10 normalized dependent variables. The basis for the manual validation is part of our replication package.[1]

Table 7
Data series in our regression analysis whose null hypothesis we rejected, thus assuming exponential growth.

Dependent variable	Absolute		Normalized	
	#cases	Share	#cases	Share
#architecture smells	6	43%	0	0%
Technical debt index	6	43%	1	7%
#affected components	8	57%	1	7%
Total order	8	57%	3	21%
Total size	8	57%	1	7%
Total #subcycles	7	50%	4	29%
Mean	7	51%	2	12%

We visualize the workflow of our regression analysis in Fig. 20. In detail, we exemplify three key variable from three different systems (rows) and show how each is transformed from the original data via a logarithmic transformation into the validation graph. The first row illustrates an example for an accepted case of exponential growth, namely the total number of subcycles per component in the system `Weka`. It is one of the few cases, in which the null hypothesis was rejected with a normalized key variable. When examining the curve of the original data series, a convex trend can be identified, while the logarithmically transformed data approximates a linear line. In the second row, we show an example (affected components in `Azureus`) for a data series that passed the requirements for a positive regression line gradient (+0.02) and sufficiently high R^2 (0.58). As its validation graph is clearly logarithmic, we did not accept exponential growth. Lastly, the null hypothesis for the technical debt index per component in the system `ANTLR` cannot be rejected, because the R^2 is only 0.2—which can be seen in the scattering of the individual data points.

We provide a summary about how many null hypotheses we rejected for each key variable in Table 7. The share of systems for which we accepted exponential growth is at least 43% for each absolute key variable, with a maximum of 57% for the number of affected components, the total order, and total size. On average, we accepted exponential growth in half of the subject systems for absolute key variables. For normalized key variables, the average is only 12%. This ranges from zero rejected null hypotheses regarding the number of architecture smells to 29% for the total number of subcycles.

Correlation Analysis. In total, 293 of the 504 data series showed positive correlations. After employing the Benjamini–Hochberg correction, we accepted 105 of these as significant positive correlations. These correlations range from a Spearman coefficient of 0.24 to 1. If we consider coefficients higher than 0.5 as a strong correlation, 93 data series exhibit a strong positive correlation. We summarize the results in Table 8, which contains the share of positive correlations for every combination of independent and dependent variables. Furthermore, we provide the mean share for every variable and the entire analysis.

The share of significant positive correlations ranges from 43% for the normalized number of architecture smells correlating with the growth of the share of affected components down to 7% for five correlations of several variables. Four of these correlations constitute the normalized total number of subcycles as the independent variable. On average, the normalized total order, size, number of architecture smells, and number of affected components correlate the most with the growth of other variables (24%). In contrast, the normalized total

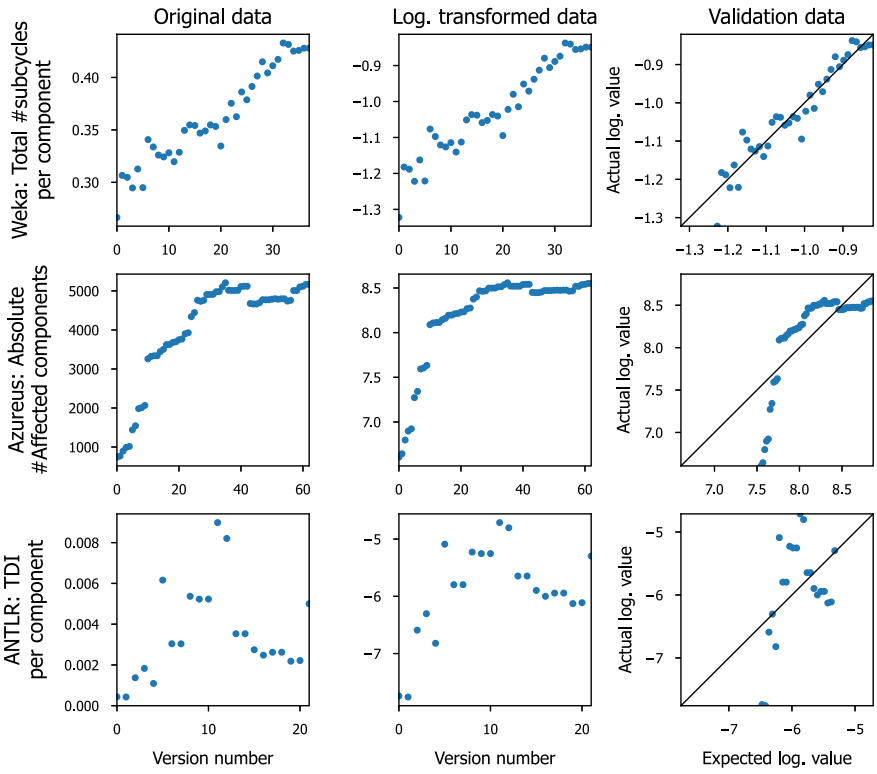

Fig. 20. Three example data series for our regression analysis.

Table 8
Share of data series in our correlation analysis whose null hypothesis we rejected, for which we thus assume a relationship. All variables are normalized according to the number of components in the respective version.

		Dependent variable, i.e., growth of...						Mean
		#ASs	TDI	#aff. comp.	Total order	Total size	Total #subcycles	
Independ. var.	#architecture smells	29%	14%	43%	21%	21%	14%	24%
	Technical debt index	7%	36%	14%	14%	14%	29%	19%
	#affected components	14%	14%	36%	29%	21%	29%	24%
	Total order	21%	14%	29%	29%	29%	21%	24%
	Total size	21%	21%	29%	29%	29%	14%	24%
	Total #subcycles	7%	21%	14%	7%	7%	7%	11%
	Mean	17%	20%	27%	21%	20%	19%	21%

RQ_3: How does the evolution of architecture smells and technical debt relate to the metaphor of compound interest?

• On average, absolute key variables on architecture smells and technical debt severity show exponential growth (i.e., compound interest) in half of the cases, while it is only 12 % for normalized variables. • Out of the latter, we accepted the highest shares of exponential growth for the total order and total number of subcycles. • High values of normalized key variables relate to higher growth rates of normalized key variables for an average of 21 % of all cases. • The growth of most key variables correlates to a similar degree with the level of various variables. •

number of subcycles is the independent variable with the lowest share of significant positive correlations (11%). Moreover, the growth of the share of affected components shows the highest share of significant positive correlations with independent variables (mean: 27%). With 17% on average, the growth of the normalized number of architecture smells is the least correlating dependent variable. Overall, 21% of all series exhibit significant positive correlations.

Discussion. We now discuss the implications we can derive from our regression and correlation analysis, respectively.

Regression Analysis. The huge difference between the shares of accepted exponential growth for absolute and normalized variables in the regression analysis shows that it is important to consider both. Additionally, given the growth of all subject systems (cf. Table 2) in the period covered in our data, we regard our hypothesis that absolute variables would often appear to grow exponentially as confirmed. It is because of the low share of normalized variables showing exponential growth that:

> *We cannot confirm that there is a general compound interest effect on architectural issues in software systems.*

This falls in line with our previous work where we analyzed the general evolution of normalized architecture smell counts and technical debt levels using descriptive statistics (Gnoyke et al., 2021).

Still, the individual results of the normalized total order and normalized total number of subcycles are interesting, because of their relatively high shares of null hypothesis rejections. For instance, the total order can increase in four ways:

1. by new architecture smells emerging in previously unaffected components,
2. by new architecture smells emerging in already affected components (i.e., a new cyclic dependency in addition to an existing hub-like dependency),
3. by existing architecture smells spreading to previously unaffected components, or
4. by existing architecture smells spreading to already affected components.

Given that we did not accept exponential growth for the normalized number of architecture smells in any system, and only one system exhibited exponential growth for the normalized number of affected components:

> *We conclude that existing architecture smells spreading to already affected components probably contributes the most to a growing total order.*

Having a closer look at the data, we can see that in several systems the share of packages that are affected by more than one architecture smell increases over time. This matches our observation in Section 5.3 that especially larger package-level cyclic dependencies and unstable dependencies tend to grow steadily over time. While there are fewer packages than classes, the share of architecture smell-affected packages is higher in most systems than the share of affected classes—balancing out the overall distribution of architecture smell types. This implies that:

> *Package-level architecture smells should not be considered less relevant than class-level architecture smells.*

Furthermore, we can conclude that:

> *Tolerating the presence of architectural issues accelerates both their own growth and the growth of adjacent architectural issues.*

We also consider the comparatively high share of exponential growth for the normalized total number of subcycles as reasonable, since few additional edges in the dependency graph of a cyclic dependency smell can cause a considerable increase in the number of subcycles. In Section 5.1, we established that merges cause significant increases in the number of subcycles, while splitting does not significantly decrease it, which is further supported by our correlation analysis. Overall, we can see another indicator that:

> *The evolution of cyclic dependencies should be closely monitored and the formation of hard-to-refactor tangling should be prevented.*

Moreover, we note that the lack of accepting exponential growth for architectural issues in a system does not imply that they did not grow at all. Even accelerating polynomial or just a constant linear growth of issues can present a challenge to developers. As we described above, more than four in five data series with a low variance ($R^2 \geq 0.5$) exhibit a regression line with a positive gradient. Ultimately, any exponential growth can only be sustained for a limited time period. In this case, if every component was affected by every architecture smell type, most key variables would reach their maximum possible value. Our correlation analysis serves as a means to better understand such general growth patterns in future research.

Correlation Analysis. The correlation analysis indicates that most independent variables have similar shares of significant positive correlations. From this, we can conclude that:

> *Our set of key variables provides a balanced view on architectural degradation and developers should focus on its entirety.*

A possible explanation for the lower share of significant positive correlations for the normalized total number of subcycles is that it is cyclic-dependency-specific. In contrast, the other key variables we investigated are influenced by all architecture smell types. Thus, we can expect a less general relationship to exist for this variable. Given our observations in Sections 5.1 and 5.2, this does not indicate that the total number of subcycles should be ignored though.

The dependent variables' means are within a confined range, too. This further affirms that all key variables are relevant to understand the evolution of a software system's architectural quality. Moreover, the "diagonal" in Table 8 is among the highest share of significant positive correlations for several dependent variables. While this short feedback loop can be expected, many variables are correlating to a similar degree with several other variables. We thus infer that:

> *Degradations in specific parts of a software system tend to affect wider parts of the system in various ways.*

This highlights the importance of regular refactoring.

In summary, we can see a higher share of significant positive correlations than significant normalized regressions (cf. Table 7). Given that we are searching for general relationships and not specifically exponential growth, this is plausible. An initially unintuitive result is the normalized total number of subcycles, whose share of significant positive correlations is considerably lower than its share of accepted exponential growth—even when correlating with its own growth. In addition to differences in how we measure growth in both cases, when looking closer at the data, we can see that eight subject systems (57%) exhibit a positive correlation for the last-mentioned combination. Six of these have a p-value below 0.14. With a higher false discovery rate, we would thus have accepted more correlations as significant.

6. Related work

In this section, we summarize related work that concerns the evolution of architecture smells, code smells, other types of smells, as well as technical debt.

6.1. Evolution of architecture smells

In our own previous work (Gnoyke et al., 2021), we first presented new conceptual techniques for analyzing the evolution of architecture smells, including new metrics, the distinction between intra- and inter-version-smells, as well as sub- and supercycles. For this article, we based on these techniques and metrics, but expanded their descriptions, presented more practical examples, novel visualizations for cyclic-dependency evolution graphs, and conducted a novel empirical study. Specifically, in our previous work, we asked how the number of architecture smells evolves over time, how architecture smells impact technical debt, and what factors influence the lifespan of architecture smells. We found that the number of architecture smells and the amount of technical debt remain mostly stable when normalizing them to the system size, but that architecture smells tend to persist in a system once they are established. Furthermore, we concluded that in different systems, different types of architecture smells contribute considerably to the technical debt despite some existing in small numbers only. In most systems, class-level cyclic dependencies contributed the most to technical debt. Lastly, we saw that class-level cyclic dependencies showed longer lifespans if they are more complex (e.g., higher order, size, number of subcycles). For unstable dependencies, longer lifespans correlated with higher centralities, overlap ratios and instability gaps, while the other types of architecture smells did not show clear patterns. So, we mostly focused on analyzing the effect of architecture smell evolution on the entire system. In contrast, in this article we studied the evolution of individual smells—especially in $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$ for which we investigated cyclic-dependency evolution graphs and how properties of architecture smells evolve over time, and

how different properties of architecture smells correlate with each other (**RQ$_3$**). We (Gnoyke et al., 2023) have also reported our experiences of developing our tool-chain by using existing tools, but this is a different topic than what we cover in this article.

Sas et al. (2019) have introduced the concept of tracking architecture smells by determining the Jaccard set similarity of component names in adjacent versions, thereby linking smell instances that we used and expanded. For cyclic dependencies, they focused on subcycles and only traced one-dimensional sequences of related intra-version smells. Sas et al. analyzed cyclic dependencies, hub-like dependencies, and unstable dependencies in the same Qualitas Corpus dataset of 14 systems and employed a toolchain of `Arcan` and their own tool `ASTracker`. They determined the normalized levels of architecture smell numbers and measured how the order, size,[5] and centrality of each smell evolved over time. For this purpose, they employed trend-evolution classification, which classified each smell into one of seven patterns, such as gradual increase or sharp decrease. This is different from our property-evolution analysis, in which we followed specific property quantiles along their increasing age and did not just determine a single label for an entire architecture-smell type. Sas et al. found the order of especially cyclic dependencies and unstable dependencies to remain mostly constant, while the size of most smells grew over time. Cyclic dependencies and hub-like dependencies showed both high shares of decreasing and increasing trends in the centrality property, while unstable dependencies mostly became less central. Given our different methods for analyzing the evolution of smell properties, we cannot directly relate the results. Nonetheless, we can conclude from our results that the order and size properties showed mostly similar trends in their distribution with increasing smell ages. Most centrality quantiles of all smell types remained either approximately constant or grew slowly over time,

In a later study Sas et al. (2022b) expanded their study to a dataset of nine C/C++ industrial closed-source systems and added the *god component* as a fourth architecture smell type. The same toolchain with `Arcan` and `ASTracker` was employed, while `Arcan` was extended to support the new language. They performed a similar survival analysis and trend evolution classification (with additional properties) as in the previous paper. In both publications, cyclic dependencies (especially on the class-level) were found to exhibit the lowest survival rates. We argue that this is due to the used definition of cyclic dependencies as subcycles, which only require a single removed dependency to be broken. Furthermore, Sas et al. analyzed how frequently different types of architecture smells co-affect the same components, whether specific types tend to precedent other types, and how cycle shapes evolve. They found cyclic dependencies to overlap the most, while god components overlapped the least. Furthermore, hub centers mostly were the center of unstable dependencies, which was very rarely the case vice-versa. Sas et al. concluded that once a cyclic dependency was introduced, other smells had a high chance to affect (some of) the same components initially, while hub-like dependencies and unstable dependencies increased the chance of cyclic dependencies and god components with increasing ages. In Section 5.2, we elaborate on the differences between our findings and those of Sas et al.. Lastly, Sas et al. interviewed 12 developers on the usefulness of architecture-smell analysis and how smells impact a system's maintainability and evolvability. The developers mostly attested that the highly architecture smell-affected parts of the systems were the most change- and issue-prone, and that architecture-smell trackers facilitate the exact identification of components that should be refactored. Commonly stated effects of architecture smells, namely ripple effects, architectural erosion, bug-proneness, and organizational challenges during development have also been confirmed by the interviewees.

[5] What we refer to as order is referred to as size by Sas et al., while our size is referred to as number of edges (cf. Section 3.4).

6.2. Evolution of code smells and design smells

Besides architecture smells, software-engineering researchers have worked extensively on code smells and their evolution. Li and Shatnawi (2007) analyzed three releases of the open-source system `Eclipse` and studied the relation of reported errors of three different severity levels to the occurrence of code smells. They focused on six smell types: (*shotgun surgery*, *god class*, *god method*, *data class*, *refused bequest*, *feature envy*). They found the former three types to occur more often in error-prone classes of all severity levels, and thus concluded that such code smells can serve to identify error-prone classes during development.

Olbrich et al. (2009) studied the evolution of the two open-source systems `Lucene` (25 aggregated versions) and `Xerces` (51 aggregated versions), regarding code smells that represent a *god class* or *shotgun surgery*. They analyzed the relative smell frequencies over time and saw alternating sequences of growing, stable, and shrinking trends. In our previous paper, we observed similar trends for architecture smells (Gnoyke et al., 2021). Also, the authors studied the change-proneness of smell-affected classes, which can be an indication of reduced maintainability. Classes that remained god classes for a prolonged time saw both increased change frequency and intensity, while for shotgun surgery-affected classes only the change frequency increased.

Vaucher et al. (2009) investigated the evolution of *god class* code smells in two open-source systems, namely `Xerxes` (36 versions) and `Eclipse JDT` (22 versions). The normalized number of smell instances remained relatively constant over time. They tracked the severity of each god class via dynamic-time warping, which showed that most instances remained constant, while the share of degrading classes was slightly higher than the share of improving classes. Only a minority of god classes seem to have been removed during the observation period. This observation is shared by several related studies we describe in this section for various smell types, including our own studies.

Chatzigeorgiou and Manakos (2010, 2013) analyzed the open-source systems `JFlex` (10 versions) and `JFreeChart` (14 versions) with respect to the number of occurrences and the lifespan of *long method*, *feature envy*, *state checking*, and *god class* code smells. They found that the number of code smells tends to increase, with many being introduced alongside their class and persisting until the end of the analyzed timespan. Removed smells were often not linked to targeted refactorings, indicating that developers did not specifically aim to remove the smells.

Peters and Zaidman (2012) studied the lifespans of five code smell types (*god class*, *feature envy*, *data class*, *message chain class*, *long parameter list class*) in three industrial and five open-source systems. They found that code smells tend to either be removed quickly after their introduction or become persistent in the system, as the rate of smell removals declines over time. Smells were often removed as side effects of maintenance activities or the implementation of new features as opposed to targeted refactoring.

Tufano et al. (2015) analyzed 200 open-source systems from which they mined half a million commits and manually examined 9164. They studied when and why the code smell types *blob class*, *class data should be private*, *complex class*, *functional decomposition*, and *spaghetti code* were introduced. The conclusions include that most classes are smell-affected since their creation, that smells are mostly introduced while implementing and enhancing features but also many times during refactoring activities, and that developers with high workloads and release pressure tend to introduce a lot of smells rather than newcomers.

Tahmid et al. (2016) studied the evolution of smell clusters of five code smell types (*god class*, *long method*, *feature envy*, *type checking*) in 10 releases of the open-source system `JUnit`. They found that the number of smell clusters (i.e., smells in the same or adjacent classes and methods) increased over time. Meanwhile, a rising share of smells was located in the same mega-cluster, as clusters merged

and the mega-cluster affected new components. On average over all versions, the mega-cluster encompassed more than two thirds of all smell-affected components. This bears similarity to how we found some cyclic dependencies to merge with each other, eventually spanning wide parts of their system. Similarly, Johannes et al. (2019) studied 12 types of code smells in 15 open-source JavaScript systems with a total of 1807 releases. They found that clean components exhibited about one third less faults than smell-affected components. Similar to the previous studies, code smells were often introduced alongside their components and persisted for a considerable time.

Habchi et al. (2019) researched eight Android-specific code-smell types in 255 thousand commits of 324 open-source Android apps. They found that some code smells remained in the code for years, while most code smells were removed after a few dozen commits. The removal rate was found to be higher for larger systems with more commits, developers, and classes. Moreover, Habchi et al. concluded that, on average, the code smell types that are detected by the static code analysis tool `Android Lint` were removed faster. Habchi et al. (2021) later extended the study by interviewing 25 Android developers and manually analyzing 561 smell-removing commits. They did not find that release pressure has an influence on the frequency of code smell introductions and removals. Code-smell removals were found to be mostly caused by large source-code removals, while refactoring was the cause in a minority of cases. Also, developers' awareness for particular smells rarely led to the smells' removal.

Rio and Brito e Abreu (2020) investigated patterns in the survival of six code-smell types (*long method*, *god class*, *long parameter list*, *deep inheritance*, *high coupling*, *high number of children*) in eight open-source PHP server-side web apps with 622 versions in total. They differentiated between localized code smells and scattered code smells, with the former being on a method or class scope and the latter affecting larger parts of the system (i.e., smelly inheritance structures). Scattered code smells were found to persist longer, but also introduced less frequently. Eventually, 60% of the observed smells were removed, while some persisted for the entire observed time period. The authors could not conclude that increasing the awareness of smells led to generally reduced survival rates.

Muse et al. (2020) analyzed 19 code-smell types and four SQL smell-types in 150 open-source data-intensive systems with a total of 1648 versions. They found that SQL smells are prevalent in the subject systems and only weakly co-occur with conventional code smells. Moreover, SQL-smell-affected components were less error-prone than code smell-affected components. Lastly, SQL smells were more often introduced alongside their component than code smells and usually remained untouched for the observation period.

Aversano et al. (2020b,a) studied the evolution of 16 design-smell types in 17,252 commits of eight open-source systems. Design smells can be categorized on a level of abstraction between code smells and architecture smells, with some overlaps in the set of considered smell types—in this case, cyclic dependencies and hub-like dependencies were included. The researchers found that smell-affected classes were more change-prone, especially when multiple smells were present in the same class. They concluded that smell removals were usually not linked to refactoring activities and that smell removals often constitute multiple smells at once.

Overall, code-smell research is an active and ongoing area, with various studies aiming to provide a better understanding of code-smell evolution. In contrast to such research, we are focusing on architecture smells, which represent software issues on a different layer of abstraction. Still, there are some overlaps regarding the methods used and findings obtained. This is not surprising and rather underpins that our results are reasonable, which also extend and complement the current state-of-the-art knowledge.

6.3. Evolution of test smells

Another type of smells that has been researched in recent years are test smells that indicate poorly designed software tests. We focus on a different kind of smells. Still, the methodologies employed and the findings provide interesting insights for architecture smells as well, and can serve to guide future research in this direction, too.

For instance, Tufano et al. (2016) performed a case study on five types of test smells in the open-source `Apache` and `Eclipse` ecosystems with a total of 190 systems and 472,116 commits. They found that test smells are mostly introduced alongside their test code and remain in the systems for a long time—which is similar to findings on code and architecture smells. The authors interviewed 19 developers and found that the interviewees did not perceive test smells as problematic. Moreover, Tufano et al. checked for relations between test smells and five types of code smells, which revealed that the presence of specific test smells often correlated with specific code smells.

Qusef et al. (2019) analyzed the evolution of 11 test-smell types in 28 versions of the open-source system `Apache Ant`. While the system grew, more test smells were observed. However, the relative level only increased very slightly with fluctuations. Several types of test smells were found to correlate with faults in production code. Similarly, Kim (2020) studied how 19 test-smell types evolved in seven open-source systems and manually analyzed commit messages in 50 commits. He found that, over time, twice as many test smells were introduced as removed. Most smell removals were linked to removing the affected code.

Later, Kim et al. (2021) performed a study on 18 test-smell types in 12 open-source systems with 102 versions in total. The researchers observed that the absolute number of test smells increased, while their relative density decreased over time. Again, most test-smell removals could be linked to inadvertent by-products of feature maintenance, while almost half of the removed test smells were found to be migrating to other test cases because of refactoring. Only a small share of test-smell removals was deliberate and concentrated on particular types of test smells.

6.4. Evolution of technical debt

Lastly, technical debt has been used in research as a common metaphor for software and systems problems stemming from postponed or sub-optimal design decisions.

Bavota and Russo (2016) studied the evolution of self-admitted technical debt in 159 open-source systems with over 600 thousand commits in total. They found that the absolute amount of technical debt increases over time alongside system growth, and that the lifespan of technical-debt artifacts is relatively long with over 1000 commits on average. Digkas et al. (2020), in turn, performed a study on 27 open-source systems in the `Apache` ecosystem with around 57 thousand commits. The authors analyzed how clean code commits can influence the density of technical debt. They observed that most revisions had a higher quality than existing code and that clean code commits strongly correlated with reducing the overall technical debt density. By analyzing meeting minutes, they found that frequently discussing code quality in board meetings led to more clean code commits.

In a related study, Digkas et al. (2017) studied 66 open-source systems in the `Apache` ecosystem with between 127 and 767 weekly snapshots each. The absolute amount of technical debt increased in the subject systems, while the relative density decreased over time in most systems. A majority of technical debt could be linked to a few specific issue types like duplicated code. Similarly, Amanatidis et al. (2017) analyzed 1564 versions of 10 open-source PHP web-application systems regarding how technical debt density in components relates to the frequency and intensity of corrective maintenance, which is a typical example of technical debt interest "payments." They confirmed a correlation in both cases.

Molnar and Motogna (2020) studied the evolution of technical debt in 110 versions of three Java open-source systems. They also found that most technical debt was caused by a small number of issue types. Technical debt fluctuated in early versions and was later mostly introduced during feature-expanding releases. The authors found technical debt reductions in versions with high refactoring activities.

Martini and Bosch (2015, 2017) focused on a related concept to compound interest in technical debt: *contagious debt*. It represents problematic parts of a system that spread to other components, risking an accelerating technical-debt growth. The authors performed a qualitative study by interviewing developers in five (later six) companies, identifying architectural technical debt items and understanding their causes as well as consequences. Using this input, they categorized sources of technical debt and mechanisms of contagious debt. Martini and Bosch concluded that the studied contagious debt did result in a compound effect. Our study is different to theirs by focusing on quantitative patterns.

7. Threats to validity

We summarize threats to validity of our study in the following. We hereby distinguish between construct, external, and internal validity.

7.1. Construct validity

We aimed to measure various aspects regarding the evolution of architecture smells. As a consequence, there are several factors that may threaten the construct validity of our work. The detected intra-version smell instances may not be complete or may represent uncritical false-positive instances. By employing the validated tool `Arcan`, which has been used in many case studies and received practitioners' feedback (Arcelli Fontana et al., 2016a; Martini et al., 2018; Sas et al., 2019, 2022b), we aimed to mitigate this threat. Also, all architecture smell types we analyzed and the software design principles they violate have been well-studied. We validated all of our modifications to `Arcan` and checked that our tools still produced the expected results. Furthermore, the matching of related intra-version smells could represent a threat to the construct validity. By performing unit tests and examining the tracking results of real systems, we made sure that our toolchain did not deviate from our concepts. One specific risk is the renaming of many components at once, which leads to false-negative tracking results, introducing seemingly high smell removal and introduction rates in the same version, as well as reducing observed smell lifespans. Solving this issue systematically (cf. Section 8) remains a future work, but we concluded that such events were rare enough in our dataset to not considerably deteriorate the results. Lastly, regarding $\mathbf{RQ_3}$, as compound interest in technical debt and software degradation is a broad field, no set of studied variables is likely to fully quantify the real world. We mitigated this issue by including a wide range of dependent and independent variables, so that, in aggregation, we can draw conclusions on our research question. Moreover, in financial debt, the principal amount, interest, and compound interest can all be quantified in the same unit of account. However, in technical debt, while the principal represents problematic system design, the interest stands for many (organizational) effects that go beyond the system. Thus, just considering increased technical debt as the compound interest is not a holistic view. This study is not meant to provide a definitive answer on the topic, but rather represent an increment to a better understanding and inspire further research.

7.2. External validity

Several external threats arise to the generalizability of this study to similar contexts. Different architecture smell types may evolve in different manners. However, as we mostly focused on particularities of our set of architecture smell types—especially cyclic dependencies, the need to generalize the observed evolution patterns is debatable. Our most general question $\mathbf{RQ_3}$ aggregates data from all studied smell types, which is why we assume that adding further smell types to the analysis would not alter the results considerably. Furthermore, the set of selected subject systems could represent a barrier for generalizing the results. While they all represent open-source Java systems, they stem from diverse domains, were developed by diverse development teams, and span many versions as well as long development periods. This way, we aimed to mitigate this threat.

7.3. Internal validity

Since we answered $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$ via qualitative analyses, we do no discuss internal validity for these. However, for $\mathbf{RQ_3}$, the question arises whether the studied relationships in the correlation and regression analysis are trustworthy and not determined by factors we did not consider. A threat to determining whether we can observe the exponential growth of technical debt is an exponentially growing system size, which we mitigated by including normalized variables in our analysis. As we studied many combinations of variables to increase construct validity, we risk observing significant trends at random. We dealt with this threat by applying the Benjamini–Hochberg multiple test correction and by being strict about accepting a scenario as significant, for example, by applying manual validations.

8. Conclusion

In this article, we advanced techniques for studying the evolution patterns of architecture smells and conducted an empirical study based on these techniques. With our empirical case study, we, first, aimed to understand the merging and splitting of cyclic dependencies, as well as how their shape (i.e., topology) evolves over time. Second, we analyzed how properties of architecture smells in general change as they manifest and evolve in a system. Lastly, we studied how the evolution of architecture smells and technical debt relate to the metaphor of compound interest (i.e., exponential growth). For this purpose, we explored 485 releases of 14 open-source Java systems, ranging up to 14 years of evolution per system and up to 485 thousand lines of code per version. We determined the set of *intra-version smells* and an extensive range of metrics in every version using a modification of the tool `Arcan`. We then used our own tool `AsTdEA` to trace architecture smells through time and to identify *inter-version smells*.

In summary, we found that cyclic-dependency merges contribute to the manifestation of very complex and hard to refactor cyclic dependencies, while splits occur less frequently and do not significantly reduce the number of subcycles—which is an indicator of proneness to change propagation in tangled cyclic dependencies. Our results on how cyclic-dependency shapes evolve support these findings: merges mostly result in the creation or growth of existing complex multi-hub cyclic dependencies, while splits often represent few components breaking away from a multi-hub without resolving the actual smell. Using our technique for visualizing inter-version cyclic dependencies as cyclic-dependency evolution graphs facilitates retracing such phenomena. Furthermore, we found complex cyclic dependencies to become even more complex as they age, while most unstable dependencies slowly grow over time, and hub-like dependencies mostly remain stable. Lastly, when looking at absolute (i.e., non-normalized) key variables of architecture-smell and technical-debt impact on software systems, we could accept exponential growth in half of the cases. Meanwhile, when normalizing said variables according to the size of the respective system version, only a minority of data series showed exponential growth. The latter also applied when observing how the severity of existing issues correlates to the growth of key variables. Overall, our results can help practitioners understand how complex architecture smells manifest themselves in their software systems. This eases coming to an informed decision where and when to efficiently

perform refactoring for avoiding such complexity from becoming unmanageable. For researchers, our contributions shed new light into the evolution of architecture smells, confirm but also question some established findings, and provide an advanced tool-base for future explorations.

As future work, we intend to transform our research methods and visualizations into new tool support for productive systems and collect feedback from developers along the way. Moreover, we want to improve the scalability of the tools' performance for large software systems. For this purpose, we aim to develop and implement concepts for detecting architecture smell-relevant differences between adjacent versions, so that repeated and redundant graph traversals can be avoided. Lastly, we intend to systematically analyze the accuracy of smell tracking to identify and implement potential improvements. For example, this could include introducing code-clone detection techniques between adjacent versions to determine the identity of two components instead of a mere name comparison. While this study focused more on cyclic dependencies than on hub-like dependencies and unstable dependencies, we intend to expand the knowledge on their evolution by analyzing how splits and mergings of their central components due to refactorings such as *extract package* or *inline class* relate to patterns in their introduction, lifespan, growth, and removal.

CRediT authorship contribution statement

Philipp Gnoyke: Writing – review & editing, Writing – original draft, Visualization, Validation, Resources, Methodology, Investigation, Data curation, Conceptualization. **Sandro Schulze:** Writing – review & editing, Validation, Methodology, Investigation, Conceptualization. **Jacob Krüger:** Writing – review & editing, Validation, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Ahmed, I., Mannan, U.A., Gopinath, R., Jensen, C., 2015. An empirical study of design degradation: How software projects get worse over time. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE.

Al-Mutawa, H.A., Dietrich, J., Marsland, S., McCartin, C., 2014. On the shape of circular dependencies in Java programs. In: Proceedings of the Australian Software Engineering Conference. ASWEC, IEEE, pp. 48–57.

Allman, E., 2012. Managing technical debt. Commun. ACM 55 (5), 50–55.

Alves, N.S.R., Mendes, T.S., de Mendonça Neto, M.G., Spínola, R.O., Shull, F., Seaman, C.B., 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121.

Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou, A., 2017. The relation between technical debt and corrective maintenance in php web applications. Inf. Softw. Technol. 90, 70–74.

Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L., 2020. Evaluating the agreement among technical debt measurement tools: Building an empirical benchmark of technical debt liabilities. Empir. Softw. Eng. 25 (5), 4161–4204.

Arcelli Fontana, F., Camilli, M., Rendina, D., Taraboi, A.G., Trubiani, C., 2023. Impact of architectural smells on software performance: an exploratory study. In: Proceedings of the International Conference on Evaluation and Assessment on Software Engineering. EASE, IEEE, pp. 22–31.

Arcelli Fontana, F., Lenarduzzi, V., Roveda, R., Taibi, D., 2019. Are architectural smells independent from code smells? An empirical study. J. Syst. Softw. 154, 139–156.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D.A., Zanoni, M., Di Nitto, E., 2017. Arcan: A tool for architectural smells detection. In: Proceedings of the International Conference on Software Architecture. ICSA, IEEE.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Zanoni, M., 2016a. Automatic detection of instability architectural smells. In: Proceedings of the International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 433–437.

Arcelli Fontana, F., Roveda, R., Zanoni, M., 2016b. Technical debt indexes provided by tools: A preliminary discussion. In: Proceedings of the International Workshop on Managing Technical Debt. MTD, IEEE, pp. 28–31.

Aversano, L., Bernardi, M.L., Cimitile, M., Iammarino, M., Romanyuk, K., 2020a. Investigating on the relationships between design smells removals and refactorings. In: Proceedings of the International Conference on Software Technologies. ICSOFT, Scite Press, pp. 212–219.

Aversano, L., Carpenito, U., Iammarino, M., 2020b. An empirical study on the evolution of design smells. Information 11 (7), 348.

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.B., 2016. Managing technical debt in software engineering. Dagstuhl Rep. 6 (4), 110–138.

Azadi, U., Arcelli Fontana, F., Taibi, D., 2019. Architectural smells detected by tools: A catalogue proposal. In: Proceedings of the International Conference on Technical Debt. TechDebt, IEEE, pp. 88–97.

Bass, L., Clements, P., Kazman, R., 2013. Software Architecture in Practice, third ed. Addison-Wesley.

Bavota, G., Russo, B., 2016. A large-scale empirical study on self-admitted technical debt. In: Proceedings of the International Conference on Mining Software Repositories. MSR, ACM, pp. 315–326.

Belady, L.A., Lehman, M.M., 1976. A model of large program development. IBM Syst. J. 15 (3), 225–252.

Benjamini, Y., Hochberg, Y., 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. J. R. Stat. Soc. Ser. B 57 (1), 289–300.

Besker, T., Martini, A., Bosch, J., 2018. Technical debt cripples software developer productivity. In: Proceedings of the International Conference on Technical Debt. TechDebt, pp. 105–114.

Besker, T., Martini, A., Bosch, J., 2019. Software developer productivity loss due to technical debt: A replication and extension study examining developer's development work. J. Syst. Softw. 156 (10), 41–61.

Cairo, A.S., de F. Carneiro, G., Monteiro, M.P., 2018. The impact of code smells on software bugs: A systematic literature review. Information 9 (11).

Cass, S., 2014. Top 10 programming languages: Spectrum's 2014 ranking. https://spectrum.ieee.org/top-10-programming-languages.

Cass, S., 2015. The 2015 top programming languages. https://spectrum.ieee.org/the-2015-top-ten-programming-languages.

Cass, S., 2016. The 2016 top programming languages. https://spectrum.ieee.org/the-2016-top-programming-languages.

Cass, S., 2017. The 2017 top programming languages. https://spectrum.ieee.org/the-2017-top-programming-languages.

Cass, S., 2018. The 2018 top programming languages. https://spectrum.ieee.org/the-2018-top-programming-languages.

Cass, S., 2019. The top programming languages 2019. https://spectrum.ieee.org/the-top-programming-languages-2019.

Cass, S., 2020. Top programming languages 2020. https://spectrum.ieee.org/top-programming-language-2020.

Cass, S., 2021. Top programming languages 2021. https://spectrum.ieee.org/top-programming-languages-2021.

Cass, S., 2022. Top programming languages 2022. https://spectrum.ieee.org/top-programming-languages-2022.

Cass, S., 2023. The top programming languages 2023. https://spectrum.ieee.org/the-top-programming-languages-2023.

Chatzigeorgiou, A., Manakos, A., 2010. Investigating the evolution of bad smells in object-oriented code. In: Proceedings of the International Conference on the Quality of Information and Communications Technology. QUATIC, IEEE, pp. 106–115.

Chatzigeorgiou, A., Manakos, A., 2013. Investigating the evolution of code smells in object-oriented systems. Innov. Syst. Softw. Eng. 10 (1), 3–18.

Cunningham, W., 1992. The WyCash portfolio management system. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications, Vol. 4, No. 2. OOPSLA, pp. 29–30.

Curtis, B., Sappidi, J., Szynkarski, A., 2012. Estimating the principal of an application's technical debt. IEEE Softw. 29 (6), 34–42.

Das, D., Maruf, A.A., Islam, R., Lambaria, N., Kim, S., Abdelfattah, A.S., Cerny, T., Frajtak, K., Bures, M., Tisnovsky, P., 2022. Technical debt resulting from architectural degradation and code smells: A systematic mapping study. ACM SIGAPP Appl. Comput. Rev. 21 (4), 20–36.

Díaz-Pace, J.A., Tommasel, A., Godoy, D., 2018. Towards anticipation of architectural smells using link prediction techniques. In: Proceedings of the Working Conference on Source Code Manipulation and Analysis. SCAM, IEEE, pp. 62–71.

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., Avgeriou, P., 2020. Can clean new code reduce technical debt density? IEEE Trans. Softw. Eng. (TSE) 48 (5), 1705–1721.

Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P., 2017. The evolution of technical debt in the apache ecosystem. In: Proceedings of the European Conference on Software Architecture. ECSA, Springer, pp. 51–66.

Fahrmeir, L., Kneib, T., Lang, S., Marx, B., 2013. Regression: Models, Methods and Applications, first ed.

Fowler, M., 2006. CodeSmell. https://martinfowler.com/bliki/CodeSmell.html.

Fowler, M., 2009. TechnicalDebtQuadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.
Fowler, M., 2019. Refactoring: Improving the Design of Existing Code, second ed. Addison-Wesley.
Gagniuc, P.A., 2017. Markov Chains: From Theory to Implementation and Experimentation, first ed.
Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009a. Identifying architectural bad smells. In: Proceedings of the European Conference on Software Maintenance and Reengineering. CSMR, IEEE, pp. 255–258.
Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009b. Toward a catalogue of architectural bad smells. In: Proceedings of the International Conference on the Quality of Software Architectures. QoSA, Springer, pp. 146–162.
Gnoyke, P., Schulze, S., Krüger, J., 2021. An evolutionary analysis of software-architecture smells. In: Proceedings of the International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 413–424.
Gnoyke, P., Schulze, S., Krüger, J., 2023. On developing and improving tools for architecture-smell tracking in java systems. In: Proceedings of the Working Conference on Source Code Manipulation and Analysis. SCAM, IEEE.
Gorton, I., 2011. Essential Software Architecture, second ed. Springer.
Gross, J.L., Yellen, J., Zhang, P., 2013. Handbook of Graph Theory, second ed. CRC Press.
Habchi, S., Moha, N., Rouvoy, R., 2021. Android code smells: From introduction to refactoring. J. Syst. Softw. 177 (7).
Habchi, S., Rouvoy, R., Moha, N., 2019. On the survival of android code smells in the wild. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 87–98.
Herold, S., 2020. An initial study on the association between architectural smells and degradation. In: Proceedings of the European Conference on Software Architecture. ECSA, Springer, pp. 193–201.
ISO/IEC 25010:2011(E), 2011. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models. Standard, International Organization for Standardization.
Izurieta, C., Bieman, J.M., 2007. How software designs decay: A pilot study of pattern evolution. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 449–451.
Johannes, D., Khomh, F., Antoniol, G., 2019. A large-scale empirical study of code smells in JavaScript projects. Softw. Qual. J. 27 (3), 1271–1314.
Karus, S., Gall, H.C., 2011. A study of language usage evolution in open source software. In: Proceedings of the International Conference on Mining Software Repositories. MSR, ACM, pp. 13–22.
Khomh, F., Di Penta, M., Guéhéneuc, Y., 2009. An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the Working Conference on Reverse Engineering. WCRE, IEEE, pp. 75–84.
Kim, D.J., 2020. An empirical study on the evolution of test smell. In: Companion Proceedings of the International Conference on Software Engineering. ICSE, ACM, pp. 149–151.
Kim, D.J., Chen, T.-H., Yang, J., 2021. The secret life of test smells – an empirical study on test smell evolution and maintenance. Empir. Softw. Eng. 26 (5).
Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. IEEE Softw. 29 (6), 18–21.
Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.-G., 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. J. Syst. Softw. 167 (9).
Laval, J., Falleri, J., Vismara, P., Ducasse, S., 2012. Efficient retrieval and ranking of undesired package cycles in large software systems. J. Object Technol. 11 (1), 260–275.
Le, D.M., Medvidovic, N., 2016. Architectural-based speculative analysis to predict bugs in a software system. In: Proceedings of the International Conference on Software Engineering. ICSE, ACM, pp. 807–810.
Letouzey, J., 2012. The SQALE method for evaluating technical debt. In: Proceedings of the International Workshop on Managing Technical Debt. MTD, IEEE, pp. 31–36.
Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101 (3), 193–220.
Li, W., Shatnawi, R., 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80, 1120–1128.
Linebach, J.A., Tesch, B.P., Kovacsiss, L.M., 2014. Nonparametric Statistics for Applied Research, first ed. Springer.
Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, first ed. Wiley.
Macia, I., Arcoverde, R., Garcia, A., Chavez, C., von Staa, A., 2012. On the relevance of code anomalies for identifying architecture degradation symptoms. In: Proceedings of the European Conference on Software Maintenance and Reengineering. CSMR, IEEE, pp. 277–286.
Martin, R.C., 1994. OO design quality metrics: An analysis of dependencies.
Martin, R.C., 2000. Design principles and design patterns. http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf.
Martini, A., Arcelli Fontana, F., Biaggi, A., Roveda, R., 2018. Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In: Proceedings of the European Conference on Software Architecture. ECSA, Springer, pp. 320–335.
Martini, A., Bosch, J., 2015. The danger of architectural technical debt: Contagious debt and vicious circles. In: Proceedings of the Working Conference on Software Architecture. WICSA, IEEE, pp. 1–10.
Martini, A., Bosch, J., 2017. On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. J. Softw.: Evol. Process 29 (10).
Melton, H., Tempero, E.D., 2007. An empirical study of cycles among classes in Java. Empir. Softw. Eng. 12 (4), 389–415.
Meyer, B., 1997. Object-Oriented Software Construction, second ed. Prentice Hall.
Mo, R., Cai, Y., Kazman, R., Xiao, L., 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: Proceedings of the Working Conference on Software Architecture. WICSA, IEEE, pp. 51–60.
Molnar, A.-J., Motogna, S., 2020. Long-term evaluation of technical debt in open-source software. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement. ESEM, ACM, pp. 1–9.
Muse, B.A., Rahman, M.M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G., 2020. On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: Proceedings of the International Conference on Mining Software Repositories. MSR, ACM, pp. 327–338.
Naik, K., Tripathy, P., 2008. Software Testing and Quality Assurance: Theory and Practice, first ed. Wiley.
Nayebi, M., Cai, Y., Kazman, R., Ruhe, G., Feng, Q., Carlson, C., Chew, F., 2019. A longitudinal study of identifying and paying down architecture debt. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, IEEE, pp. 171–180.
Nixon, M.S., Aguado, A.S., 2002. Feature Extraction and Image Processing, first ed. Newnes.
Olbrich, S.M., Cruzes, D.S., Basili, V.R., Zazworka, N., 2009. The evolution and impact of code smells: A case study of two open source systems. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 390–400.
Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.K., 2010. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: Proceedings of the International Conference on Software Maintenance. ICSM, IEEE, pp. 1–10.
Parnas, D.L., 1994. Software aging. In: Proceedings of the International Conference on Software Engineering. ICSE, IEEE, pp. 279–287.
Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: Proceedings of the European Conference on Software Maintenance and Reengineering. CSMR, IEEE, pp. 411–416.
Power, K., 2013. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options. In: Proceedings of the International Workshop on Managing Technical Debt. MTD, IEEE, pp. 28–31.
Qusef, A., Elish, M.O., Binkley, D., 2019. An exploratory study of the relationship between software test smells and fault-proneness. IEEE Access 7.
Rachow, P., Riebisch, M., 2022. An architecture smell knowledge base for managing architecture technical debt. In: Proceedings of the International Conference on Technical Debt. TechDebt, pp. 1–10.
Rangnau, T., 2020. Determining the Rationale of Architectural Smells from Issue Trackers (Master's thesis). University of Groningen.
Rio, A., Brito e Abreu, F., 2020. PHP code smells in web apps: Survival and anomalies. https://arxiv.org/abs/2101.00090v1.
Rizzi, L., Arcelli Fontana, F., Roveda, R., 2018. Support for architectural smell refactoring. In: Proceedings of the International Workshop on Refactoring Tools. WRT, ACM, pp. 7–10.
Roveda, R., 2018. Identifying and Evaluating Software Architecture Erosion (Ph.D. thesis). University of Milano-Bicocca.
Roveda, R., Arcelli Fontana, F., Pigazzini, I., Zanoni, M., 2018a. Towards an architectural debt index. In: Proceedings of the International Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 408–416.
Roveda, R., Arcelli Fontana, F., Pigazzini, I., Zanoni, M., 2018b. Towards an architectural debt index. In: Proceedings of the International Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 408–416.
Sangwan, R.S., Vercellone-Smith, P., Laplante, P.A., 2008. Structural epochs in the complexity of software over time. IEEE Softw. 25 (4), 66–73.
Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., do Nascimento, R.S., Freitas, M.F., de Mendonça, M.G., 2018. A systematic review on the code smell effect. J. Syst. Softw. 144, 450–477.
Sas, D., Avgeriou, P., 2023. An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells. IEEE Trans. Softw. Eng. (TSE) 49 (8), 4169–4195.
Sas, D., Avgeriou, P., Arcelli Fontana, F., 2019. Investigating instability architectural smells evolution: An exploratory case study. In: Proceedings of the International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 557–567.
Sas, D., Avgeriou, P., Pigazzini, I., Arcelli Fontana, F., 2022a. On the relation between architectural smells and source code changes. J. Softw.: Evol. Process 34 (1).
Sas, D., Avgeriou, P., Uyumaz, U., 2022b. On the evolution and impact of architectural smells: An industrial case study. Empir. Softw. Eng. 27 (4).
Sedgewick, R., Wayne, K., 2011. Algorithms, fourth ed. Addison-Wesley.
Sharma, T., Singh, P., Spinellis, D., 2020. An empirical investigation on the relationship between design and architecture smells. Empir. Softw. Eng. 25 (5), 4020–4068.

Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T., 2012. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. (TSE) 39 (8), 1144–1156.

Spearman, C., 1904. The proof and measurement of association between two things. Am. J. Psychol. 15 (1), 72–101.

Stevens, W.P., Myers, G.J., Constantine, L.L., 1974. Structured design. IBM Syst. J. 13 (2), 115–139.

Sugiyama, K., Tagawa, S., Toda, M., 1981. Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybern. 11 (2), 109–125.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt, first ed. Elsevier.

Tahmid, A., Nahar, N., Sakib, K., 2016. Understanding the evolution of code smells by observing code smell clusters. In: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 8–11.

Tarjan, R.E., 1972. Depth-first search and linear graph algorithms. SIAM J. Comput. 1 (2), 146–160.

Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2010. Software Architecture: Foundations, Theory, and Practice, first ed. Wiley.

Tempero, E.D., 2013. Qualitas corpus index: Release 20130901 (evolution distribution). http://qualitascorpus.com/docs/catalogue/20130901/corpus_catalogue-evolution.html.

Tempero, E.D., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010. The Qualitas Corpus: A curated collection of Java code for empirical studies. In: Proceedings of the Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 336–345.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells. In: Proceedings of the International Conference on Automated Software Engineering. ASE, ACM, pp. 4–15.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and why your code starts to smell bad. In: Proceedings of the International Conference on Software Engineering. ICSE, IEEE, pp. 403–414.

Vaucher, S., Khomh, F., Moha, N., Guéhéneuc, Y.-G., 2009. Tracking design smells: Lessons from a study of god classes. In: Proceedings of the Working Conference on Reverse Engineering. WCRE, IEEE, pp. 145–154.

Verdecchia, R., Malavolta, I., Lago, P., 2018. Architectural technical debt identification: The research landscape. In: Proceedings of the International Conference on Technical Debt. TechDebt, ACM, pp. 11–20.

Vidal, S.A., Guimarães, E.T., Oizumi, W.N., Garcia, A., Pace, J.A.D., Marcos, C.A., 2016. Identifying architectural problems through prioritization of code smells. In: Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse. SBCARS, IEEE, pp. 41–50.

Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q., 2016. Identifying and quantifying architectural debt. In: Proceedings of the International Conference on Software Engineering. ICSE, ACM, pp. 488–498.

Yin, R.K., 2018. Case study research and applications: Design and methods. Sage.

Zazworka, N., Shaw, M.A., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Proceedings of the International Workshop on Managing Technical Debt. MTD, ACM, pp. 17–23.

Zazworka, N., Vetro', A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., Shull, F., 2014. Comparing four approaches for technical debt identification. Softw. Qual. J. 22 (3), 403–426.

Zhang, M., Hall, T., Baddoo, N., 2011. Code bad smells: A review of current knowledge. J. Softw. Maint. Evol. 23 (3), 179–202.

von Zitzewitz, A., 2019. Mitigating technical and architectural debt with sonargraph: Using static analysis to enforce architectural constraints. In: Proceedings of the International Conference on Technical Debt. TechDebt, IEEE, pp. 66–67.