The Journal of Systems and Software 225 (2025) 112382

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

An empirical investigation of the impact of architectural smells on software maintainability[☆]

Rodi Jolak[*], Simon Karlsson, Felix Dobslaw

Mid Sweden University, Sweden

ARTICLE INFO

Keywords:
Software architecture
Architectural smells
Software maintenance
Software quality
Software metrics

ABSTRACT

In recent years, interest in the potential influence of architectural smells on software maintainability has grown. Yet, empirical evidence directly linking maintainability quality attributes — such as modularity and testability — with architectural smells is scarce. This study analyzes seven architectural smells across 378 versions of eight open-source projects. We developed a tool to gather data on architectural smells, associated projects, and package-level quality attributes. The empirical findings reinforce that certain architectural smells indeed correlate with specific quality attributes - a notion previously backed merely by argument. Most architectural smells negatively correlated with project-level testability but not project-level modularity. While there is not a consistent negative correlation with testability at the package level, many smells display a pronounced negative association with the maintainability quality attributes.

1. Introduction

Architectural smells are recurring, identifiable architectural design decisions that mainly impact the maintainability of a system (Rachow and Riebisch, 2022; Tian et al., 2019; Martini et al., 2018; Sas et al., 2022). According to Lippert and Roock (2006), architectural smells occur when recognized design principles such as *single-responsibility*, *open-closedness*, or *composability* are violated. Architecture smells are indicators of potential technical debt, a concept introduced by Cunningham in 1992 (Cunningham, 1992). It suggests that using "immature" code may speed up the development process but at the cost of incurring a "debt" that would eventually need to be repaid through rewrites. Technical debt can impede a system's evolvability and maintainability (Kruchten et al., 2012). Architectural debt is only one type of technical debt. Baabad et al. (2020) identify several concepts of architectural deviation, including architectural degeneration, erosion, drift, mismatch, decay, degradation, and more. Further, Stochel et al. (2020) conclude that there is ambiguity in how the various terms are applied in research.

Architectural smells are conceptually similar to code smells, but they operate at different levels. While code smells focus on the implementation level, such as methods, classes, parameters, and statements, architectural smells target the architecture level, including components, connectors, interfaces, and configurations (Garcia et al., 2009a). In a study comparing the two, Arcelli Fontana et al. (2019) found no correlation[1] between code smells and architectural smells and that addressing one did not affect the other. They conclude that architectural smells are "independent from code smells, and therefore deserve special attention by researchers, who should investigate their actual harmfulness". However, further studies examining the statistical relationship (i.e., correlation) between architectural smells and code smells are required since this relationship can vary depending on the context and the specific smells being studied.

Recent studies show that architectural smells are prevalent in open-source projects. In a study spanning over 86,000 Java and C# GitHub repositories, Sharma and Kessentini (2021) found nearly 1.2 million instances of seven distinct types of architectural smells. Meanwhile, Arcelli Fontana et al. (2019) observed at least two architectural smell types in almost all (102/103) open-source projects they investigated. Notably, they only considered a total of three architectural smells.

Architectural smells are understood to impact software quality attributes negatively. According to the ISO/IEC 25010:2023 standard (ISO/IEC 25010:2023, 2023), software quality has two distinct facets: quality in use and product quality. While quality in use is concerned with how a system meets the needs of its users during actual use, product quality is concerned with the design and operation of the software that enables and supports its use. Architectural smells are

[☆] Editor: Neil Ernst.

[*] Corresponding author.

E-mail addresses: rodi.jolak@miun.se (R. Jolak), sika2001@student.miun.se (S. Karlsson), felix.dobslaw@miun.se (F. Dobslaw).

URL: https://www.rodijolak.com (R. Jolak).

[1] "Correlation" is a statistical technique to examine the "relationship" between two variables (Lee Rodgers and Nicewander, 1988).

https://doi.org/10.1016/j.jss.2025.112382
Received 3 June 2024; Received in revised form 1 January 2025; Accepted 9 February 2025
Available online 20 February 2025

concerned with the latter. Source code metrics are needed but can be hard to capture (Mehboob et al., 2021).

1.1. Motivation

The assertion that architectural smells negatively impact quality has been criticized as based on individual developer experience and intuition rather than empirical data (Rachow and Riebisch, 2022; Le et al., 2018). Some recent studies show that certain architectural smells can negatively impact the number of issues and changes in open-source projects (Sas et al., 2022; Le et al., 2018). However, other recent studies show that sometimes architectural smells are part of a deliberate solution (Martini et al., 2018; Pigazzini et al., 2021a). The assertion that architectural smells themselves are the underlying cause of quality degradation has also been questioned, as co-changes in files were found to appear before any smell was detected (Sas et al., 2021).

Detecting smells may impact the development of a project in terms of the allocation of time and resources to fix them. Additionally, refactoring architectural smells can be nontrivial, with a risk of introducing other smells in the process (Chantian and Muenchaisri, 2019). Thus, there is a need to measure the impact of individual smells to help developers make informed decisions. Therefore, this work aims to empirically investigate the impact of architectural smells on maintainability.

Maintainability refers to the efficiency and effectiveness with which a system can be modified by its intended maintainers, including error correction, improvements, adaptations, updates, and upgrades (ISO/IEC 25010:2023, 2023). Maintainability comprises several sub-characteristics,[2] and individual architectural smells are inferred to impair certain sub-characteristics based on violated design principles (Rachow and Riebisch, 2022). For instance, the Cyclic Dependency architectural smell is claimed to impair all sub-characteristics of maintainability and is therefore thought to be one of the most critical smells (Gnoyke et al., 2021; Rachow and Riebisch, 2022). Here, components can be classes, packages, or even whole subsystems (Lippert and Roock, 2006). Because of transitivity, this dependency can be hard to detect. Lippert and Roock (Lippert and Roock, 2006) explain that cyclic dependencies can affect maintainability negatively since they can have "severe and unpredictable consequences". Modifying one part of the cycle can have side-effects in any other part of the cycle (Lippert and Roock, 2006).

In a systematic mapping study on architectural smell detection, Mumtaz et al. (2021) stress the importance of investigating the sub-characteristics of maintainability. They suggest that particularly testability has received limited attention in research thus far.

1.2. Contribution

In this study, we examine various open-source software projects and statistically assess the relationship between architectural smells and maintainability quality attributes, specifically modularity and testability. Modularity refers to the degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components. Low modularity, characterized by high coupling, low cohesion, or poor separation of concerns, can increase the number of unnecessary changes to neighboring components. Testability refers to the degree of effectiveness and efficiency with which test criteria can be established, and tests can be performed to determine whether those criteria have been met for a system or component.

We investigate modularity and testability because quantifiable means exist to assess them. Modularity is measured using Decoupling Level and Propagation Cost (Mo et al., 2018). Considering testability, although not all aspects of testability are easily quantifiable, the effectiveness and efficiency of establishing test criteria for a system can be estimated by measuring the test coverage of files impacted by architectural smells and comparing these metrics to a baseline Other maintainability sub-characteristics are not easily nor accurately measurable. For instance, for reusability, several metrics have been proposed. However, factors influencing reusability were not easy to quantify. Existing measures instead depend on proxy metrics that do not present any correlation concerning reusability (Mehboob et al., 2021).

This study addresses the following research questions:

RQ1. *What is the relationship between architectural smells and (a) Project-level modularity, (b) Project-level testability, (c) Testability in packages?* The purpose of RQ1 is to establish whether there is a statistical relationship (i.e., correlation) between the measured architectural smells and quality attributes. Our study specifically focused on architectural smells, which are concerned with the architecture level, including components, connectors, interfaces, and configurations. These aspects are best analyzed at the *project level* to understand the overall modularity and structural integrity of the system. Class-level modularity can also influence system structure. Therefore, a future research exploring the impact of class-level modularity on system architecture is needed to provide a more comprehensive understanding of this relationship.

RQ2. *How do relationships compare between different types of architectural smells and (a) Project-level modularity, (b) Project-level testability, (c) Testability in packages?* RQ2 looks at each measured type of architectural smell individually to see how they statistically relate (i.e., correlate) to the measured quality attributes. The purpose is to examine whether the architectural smells that are said to impair modularity or testability exhibit stronger or weaker statistical relationships with these quality attributes.

The contribution of this study is two-fold: First, we introduce an open-source Architectural Smell Analysis Tool (ASAT) that supports gathering and analyzing data related to architectural smells, modularity, and/or testability from multiple versions of open-source projects hosted on GitHub. Second, we provide empirical evidence highlighting a clear distinction among different types of architectural smells concerning their statistical relationship with modularity and testability.

1.3. Structure of the paper

The remainder of the paper follows a standard structure with sections on Related Work 2, Research Methodology 3, Results 4, Discussion 5, and finally concluding Remarks 6.

2. Related work

In this section, we first present work related to architectural technical debt. Second, we report work related to architectural smells. Third, we report work related to empirical studies on architectural smells. Last, we outline how our work differs from the related work.

2.1. Architectural technical debt

Verdecchia et al. (2020) investigated how software development organizations perceive and manage their architectural technical debt. They used a grounded theory approach to gather qualitative data from software architects and senior technical staff across various organizations. The resulting conceptual theory of architectural technical debt encompasses critical concepts such as symptoms, causes, consequences, and management strategies. This theory, grounded in empirical data, provides valuable insights into the critical factors influencing architectural technical debt in industrial contexts.

Verdecchia et al. (2018) conducted a systematic mapping study methodology to identify, classify, and evaluate the current state of the

[2] The sub-characteristics of maintainability according to ISO/IEC 25010:2023 (ISO/IEC 25010:2023, 2023) are modularity, reusability, analyzability, modifiability, and testability.

art in architectural technical debt identification. The study examines publication trends, the characteristics of different approaches, and their potential for industrial adoption. From an initial pool of 509 potentially relevant studies, 47 primary studies were systematically selected and analyzed using a rigorously defined classification framework. The study offers an assessment of current research trends and gaps in architectural and technical debt identification, a solid foundation for understanding existing and future research, and a rigorous evaluation of the potential for industrial adoption of identification methods of architectural and technical debt.

2.2. Architectural smells

The first architectural smells were defined by Lippert and Roock (2006), which included the Cyclic Dependency smell that has been cited as the most impactful smell (Gnoyke et al., 2021; Rachow and Riebisch, 2022). Since these smells operate on the architectural level, they involve components such as classes, packages, and subsystems, requiring larger refactorings to be handled (Lippert and Roock, 2006). Architectural smells are characterized as violating design principles (Lippert and Roock, 2006; Rachow and Riebisch, 2022; Azadi et al., 2019), and reported to negatively impacting the maintainability of systems (Garcia et al., 2009a; Rachow and Riebisch, 2022; Tian et al., 2019; Martini et al., 2018; Sas et al., 2022) for which there is a lack of empirical studies. It has, though, been argued that not all smells are problematic and need to be analyzed on a case-by-case basis (Lippert and Roock, 2006; Pigazzini et al., 2021b). Thus, in some situations, false positives must be considered (Martini et al., 2018; Pigazzini et al., 2021b). Garcia et al. (2009a) further defined four architectural smells and characterized architectural smells as a commonly used architectural decision that negatively impacts system quality, intentional or not.

Numerous other architectural smells have been identified. In a systematic mapping study, Mumtaz et al. (2021) found and described 108 architectural smells. However, existing tools or techniques do not detect many of these smells.

Azadi et al. (2019) presented a catalog of 12 architectural smells that are detectable by current tools, classified according to violated design principles. They propose a classification of the smells according to the violation of three design principles: modularity, hierarchy, and healthy dependency structure.

Rachow and Riebisch (2022) conducted a systematic literature review to look for explicit connections between identified architectural smells and quality attributes. They identified 132 connections between architectural smells and maintainability or one of its sub-characteristics and 139 connections to violations of design principles. Because design principles promote quality, they argued that violating a principle should impair the associated quality attributes. The results were used to compile a knowledge base of 114 architectural smells with links to the sub-characteristics of maintainability that they were found to impair However, the authors acknowledged that, in general, the connections between architectural smells and quality attributes or design principles were merely asserted or supported by an argument rather than being demonstrated empirically.

Le et al. (2016) presented a taxonomy of architectural smells, associated metrics, and their impact on quality properties. They relate architectural smells to maintenance and evolution areas and thus provide a foundational step towards a comprehensive approach to estimating system sustainability. Initial results from a set of subject systems demonstrate the potential of the taxonomy to guide future research and practical applications in maintaining the architectural integrity of software systems.

Fontana et al. (2016) created an open-source tool called Arcan to detect architectural smells that may compromise system stability. This tool identifies three architectural smells: unstable dependency, hub-like dependency, and cyclic dependency. The detection technique of the Arcan tool exploit graph database technology, allowing for high scalability in smell detection and better management of large amounts of dependencies of multiple kinds.

2.3. Empirical studies on architectural smells

Gnoyke et al. (2021) conducted a study on the evolution of three architectural smells across 485 versions of 14 open-source projects. They found that although the number of smells tended to increase with the size of the system, the relative amount of smells mainly remained stable. They suggest that smells may be caused more by the sheer amount of source code than by specific architectural properties, although they note that further research is needed to support this claim. However, they emphasize the need to first examine to what extent architectural smells pose an actual problem.

Le et al. (2018) aimed to provide empirical evidence of the impact of architectural smells on software maintainability. They examined reported issues and the number of commits as proxies for maintainability and compared them to six detected architectural smells across 421 versions of eight open-source projects. The results revealed that files affected by architectural smells not only had more reported issues but also more commits, particularly in files with long-lived smells, where both metrics increased over time.

Sharma et al. (2020) investigated the characteristics of architecture smells, and their relationships with design smell. They implemented detection support for seven architecture smells and analyzed over 3073 open-source repositories providing empirical evidence on the correlation, collocation, and causation between architecture and design smells. The findings reveal that while architecture smells strongly correlate with design smells, they do not typically collocate, and design smells often precede architecture smells, suggesting a causal relationship.

Capilla et al. (2023) explored how architectural smells contribute to architectural debt in microservices. Using tools like Arcan and Designite, the study analyzes 20 open-source projects to explore the correlation between architectural smells and microservices. The findings underscore the critical role of architectural smells in recognizing architectural debt, providing empirical data that can guide practitioners in identifying and mitigating technical debt in microservice architectures.

Sas et al. (2022) also used source code changes to explore the impact of four architectural smells on maintainability. They measured change frequency and change size as proxies for maintenance effort and analyzed commits taken at 4-week intervals in 31 open-source Java systems. The results showed that components affected by architectural smells change more frequently and extensively. They also found that as the number of smells that affect a component increases, so does the likelihood that it will change. The type of architectural smell did not have a significant correlation with changes. Still, they note that potentially not all types of cycles impact changes, which is consistent with previous research (Lippert and Roock, 2006; Pigazzini et al., 2021b; Martini et al., 2018). However, they found that introducing a smell only increased the changes in the affected component in some cases—sometimes, the opposite was true.

2.4. Comparison with related work

Similar to the aforementioned empirical studies, this study will measure architecture smells in several open-source projects over time. It will employ a methodology inspired by Sas et al. (2022) for project selection and using a fixed interval of versions for each project. However, this study will measure seven architectural smells—a greater number than prior studies. Additionally, previous studies used various metrics as proxies for technical debt or maintainability, whereas this study will aim to measure sub-characteristics of maintainability, specifically modularity and testability. To our knowledge, no studies have been conducted on the correlation between architectural smells and sub-characteristics of maintainability, even though it has been identified as an essential subject for future research (Mumtaz et al., 2021). Furthermore, this will help verify the links between individual architectural smells and these sub-characteristics in the knowledge base by Rachow and Riebisch (2022), which currently lacks supporting empirical evidence.

3. Research methodology

The study is conducted as an embedded multiple case study (Yin, 2018). Each case consists of an open-source Java project with a chronological sequence of units of analysis—the commit-based versions of the project at a given point in its development.

3.1. Propositions

To guide the case study, we propose several theoretical propositions that highlight the key aspects to be explored and provide a structured approach for analyzing the study's findings.

- Proposition A: Architectural smells are often described as impacting maintainability negatively (Rachow and Riebisch, 2022; Tian et al., 2019; Martini et al., 2018; Sas et al., 2022). Therefore, we propose that the statistical relationship between architectural smells and the sub-characteristics of maintainability (testability and modularity) is negative. If our findings do not support this proposition, then a deeper investigation is needed to understand the deviations from the expected negative relationship.
- Proposition B: Previous studies found a statistical relationship in components between architectural smells and changes and issues (Le et al., 2018; Sas et al., 2022), which were used as proxies for maintainability. Since testability is a sub-characteristic of maintainability, it is reasonable to assume that testability in packages should also have a negative statistical relationship with architectural smells. Therefore, we propose that there is a negative statistical relationship between architectural smells in individual packages and the testability of those packages.
- Proposition C: Rachow and Riebisch (2022) introduced a knowledge base of architectural smells and the maintainability sub-characteristics they supposedly impair. Their claims are founded on violated design principles and suggested connections between architectural smells and quality attributes, as discussed in research based on theoretical foundations.

  Particularly, according to Rachow and Riebisch (2022) the following architectural smells are considered to impair modularity (i.e., modularity impairing group): Ambiguous Interface, Cyclic Dependency, Dense Structure, God Component, and Scattered Functionality. In contrast, the smells that are considered by Rachow and Riebisch (2022) to non-impair modularity (i.e., modularity non-impairing group) are: Feature Concentration and Unstable Dependency.

  Considering testability, the following smells are considered by Rachow and Riebisch (2022) to impair testability (i.e., testability impairing group): Dense Structure, God Component, and Scattered Functionality. In contrast, the smells that are considered by Rachow and Riebisch (2022) to non-impair testability (i.e., testability non-impairing group) are: Ambiguous Interface, Cyclic Dependency, Feature Concentration, and Unstable Dependency.

  Based on this, certain architectural smells (i.e., impairing group) should have a stronger negative statistical relationship with associated sub-characteristics of maintainability. Accordingly, we propose that the types of architectural smells that are said to specifically impair modularity and/or testability exhibit a stronger negative statistical relationship with these attributes compared to the non-impairing smells. If our findings do not support this proposition, then a deeper investigation is needed to understand the deviations from the expected relationship.
- Proposition D: Gnoyke et al. (2021) theorized that architectural smells might be a symptom of the size of a project rather than any architectural properties. As a rival explanation, size may be a *confounding factor* that explains the observed relationships between architectural smells and modularity/testability, whereby size explains both the increase in architectural smells and the decrease in modularity/testability. Here, we want to provide a more comprehensive understanding of the confounding factor (i.e., project size) affecting modularity and testability, as well as ensure that our conclusions are well-founded and actionable. Accordingly, we propose that the statistical relationship between size and architectural smells is *stronger* than that between architectural smells and modularity/testability. Additionally, the statistical relationship between size and modularity/testability is *stronger* than that between architectural smells and modularity/testability.

3.2. Cases

To ensure a representative sample of open-source projects, eight projects (see Table 1) are selected from GitHub based on the following inclusion criteria, inspired by Sas et al. (2022). The project shall

- be nontrivial and has a size of at least 10,000 lines of code in its latest commit.
- contain at least one instance of an architectural smell in its latest commit.
- be in active development for at least three years.
- show consistent activity during development.[3]
- be made to build successfully.

We further decided to limit the study to projects developed in Java to manage tooling complexity while adhering to a popular programming language that has been applied to a wide range of use cases. The filtered GitHub projects were analyzed over their entire lifetime, providing 378 versions, roughly totaling 29 years of active development data.

3.3. Data collection and analysis

For each project, versions are taken at 4-week intervals from project inception to the latest commit as used by Sas et al. (2022), where it produced a meaningful change in calculated metrics between versions. Further, fixed intervals lead to consistent results. Shorter intervals provide greater granularity at the cost of greater processing time and result in project variance.

Designite[4] is used to detect the architectural smells: Ambiguous Interface, Cyclic Dependency, Feature Concentration, God Component, Scattered Functionality, Unstable Dependency, and Dense Structure. Designite is used since it offers an academic license. It detected more smells than other tools, and its detection methods are evaluated by Azadi et al. (2019).

According to Sharma et al. (2020), the detection mechanism of the aforementioned architectural smells is done via the Designite tool as follows:

- *Ambiguous Interface*: This smell arises when a component provides a single general entry-point into a component (Garcia et al., 2009b). This smell is detected by identifying components that are not too small (i.e., have at least five classes) and contain only one public or internal method.
- *Cyclic Dependency*: This smell arises when two or more components depend on each other either directly or indirectly (Mo et al., 2015). For each component, a dependency list is computed where component A depends on component B if at least one class in component A refers to at least one class in component B. Based on this, a directed graph is created, and a depth-first search algorithm is applied to detect cycles in the graph for each component.

[3] As shown by the Contributors page on GitHub.

[4] Designite can be found here: https://designite-tools.com.

Table 1
Projects Selected for Analysis (RPC stands for Remote Procedure Call). Some projects do not have any code or smells in the first version (reported as zero), so we report the LOC and total number of smells in the second version instead, in parentheses.

Description	Project name	1st Version	LOC in 1st (2nd) version	Architectural Smells in 1st (2nd) version	Versions	Unique Packages	LOC in Last Version	Classes in Last Version	Architectural Smells in Last Version
Microservices framework	light-4j	2016–10–09	0 (4843)	0 (14)	76	115	68k	906	101
RPC framework	motan	2016–04–20	18 903	75	61	76	43k	621	108
Performance monitoring tool	MyPerf4J	2018–11–04	0 (2733)	0 (7)	33	118	17k	322	56
Distributed transaction solution	seata	2019–02–10	22 594	85	51	372	238k	2527	405
Flow control component	Sentinel	2018–08–24	18 051	79	50	366	181k	1516	354
RPC framework	sofa-rpc	2018–06–08	0 (45091)	0 (101)	46	186	84k	1239	189
Validation framework	yavi	2018–08–21	1150	3	33	18	20k	533	40
Java library	zip4j	2019–07–04	7761	38	28	17	15k	150	56

- *Feature Concentration*: This smell occurs when a component realizes more than one architectural concern or feature (i.e., a not cohesive component) (De Andrade et al., 2014). Designite computes the Lack of Component Cohesion (LCC) to measure the component cohesion. Related classes in a component are identified to compute LCC, a dependency graph is created, and then the number of disconnected sub-graphs is identified. Two classes are related if they share association, aggregation, composition, or inheritance relationships.
$$LCC = \frac{\text{Number of disconnected sub-graphs}}{\text{Total number of classes}}$$
The smell is detected if LCC exceeds a pre-defined threshold of 0.2. This threshold is adopted after experimenting with various threshold values and manually analyzing the resultant set of detected smells (Sharma et al., 2020).
- *God Component*: This smell arises when a component is large either in terms of lines of code or number of classes (Lippert and Roock, 2006). Designite detects god component architecture smell when a component has more than 30 classes or 27000 lines of code (Lippert and Roock, 2006).
- *Scattered Functionality*: This indicates that multiple components are responsible for realizing the same architectural high-level concern (Garcia et al., 2009b). Designite monitors how often a method accesses external namespaces together. Frequent occurrences of such accesses within a component can result in a scattered functionality architecture smell in the affected components.
- *Unstable Dependency*: The Stable Dependencies Principle (SDP) (Martin, 2003) asserts that dependencies between packages should align with their stability. Therefore, a package should only depend on packages that are more stable than itself. An unstable dependency architecture smell arises when this principle is violated. The instability of a component is computed as follows:
$$I = \frac{C_e}{C_e + C_a}$$
Where I represent the degree of instability of the component; C_a represents the afferent coupling (i.e., incoming dependencies), and C_e represents the efferent coupling (i.e., outgoing dependencies).
- *Dense Structure*: This smell occurs when the components form a very dense dependency graph (Sharma et al., 2016). Designite forms a dependency graph among all the components and computes the average degree of the graph as follows:
$$Average\ Degree = \frac{2 * |E|}{|V|}$$
E is the set of all the edges among the vertexes, and V is the set of all vertexes belonging to the graph. The default threshold Designite uses to identify dense structure is average degree >= 5.

In addition, metrics relating to modularity and testability are measured. Regarding modularity, Propagation Cost and Decoupling Level were measured using the tool DV8.[5] DV8 provides an academic license, and it has been used since it was possible to integrate it into our tool pipeline.

Decoupling Level and Propagation Cost are selected since they are applied on projects in practice with favorable results. Mo et al. in Mo et al. (2018) verified that these metrics capture software architects' sentiments about the maintainability of projects in the study.

Decoupling Level (Mo et al., 2016) captures how well a system is divided into discrete modules. The calculation formula for the Decoupling Level (DL) is the following:

$$DL = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{C_i} \right)$$

Where:

- N is the total number of elements in the system.
- C_i is the degree of component i, representing the number of dependencies that component i has.

This formula averages the inverse of the degrees of all components, indicating how decoupled the system is.

Propagation Cost (MacCormack et al., 2006) measures how tightly coupled a system is. The calculation formula of the Propagation Cost (PC) is the following:

$$PC = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{N} \sum_{j=1}^{N} M_{ij}$$

Where:

- N is the total number of elements in the system.
- M_{ij} is an element in the *design structure matrix* representation of the system's dependencies, which is 1 if there is a dependency from element ‘i’ to element ‘j’, and 0 otherwise.

Testability is the degree of effectiveness and efficiency with which test criteria can be established, and tests can be performed to determine whether those criteria have been met (ISO/IEC 25010:2023, 2023). JaCoCo[6] is used to assess testability based on code coverage in terms of line-coverage executed by tests. JaCoCo is open source and used since it provides detailed metrics on line coverage, branch coverage, and complexity, which help understand how much of the code is being tested. Moreover, it can be integrated seamlessly with popular build tools like Gradle, making it easy to incorporate into our tool pipeline.

While architectural smells and modularity characteristics could be assessed using static analysis, testability, therefore, required dynamic analysis, which required project versions to build successfully. Build failures are exceedingly common with their leading root cause being test failures (Yasi and Qin, 2022). Thus, project selection was heavily influenced by the ability to build a project (in Section 3.2). Flaky and unreliable[7] projects were excluded, as further discussed in the threats section. In addition, even for stable projects, certain versions would not build and were excluded too. The identification and exclusion of flaky and unreliable projects included the following steps:

- Projects needed to successfully pass 90% of their tests across all versions during the building phase.
- We excluded projects that showed a large variance in test failures from one build to another.
- If the ratio of successful tests had varied wildly or tests were failing randomly without apparent explanation, that data would have been deemed unreliable and the project excluded.
- Certain versions of a project are excluded if they would not build or there is a sudden discrepancy in the build. For example, the project yavi obtained 100% test success rate except a single early version which only had 87% test success. Including that version might influence the results for unrelated reasons since it was not clear what was causing the test failures

[5] DV8 can be found here: https://archdia.com/.

[6] JaCoCo can be found here: https://www.jacoco.org/jacoco/trunk/index.html.

[7] Refers to unsuccessful tests or sudden fluctuation in test success rate between versions when building projects.

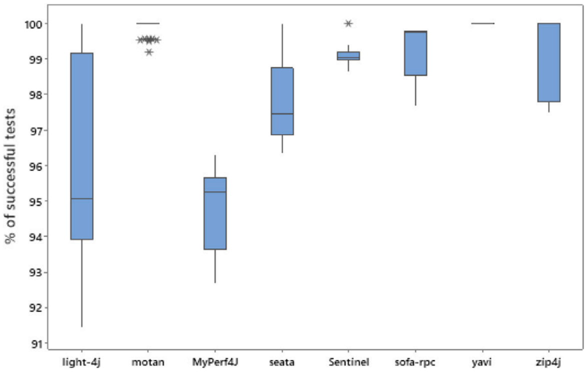

Fig. 1. Successful tests over all versions per project.

The boxplot in Fig. 1 displays the overall test success rate across all included versions of the selected projects.

Project-size has been identified as a potential confounding variable, potentially affecting the occurrence of architectural smells (Gnoyke et al., 2021). The number of classes may even impact Propagation Cost (Mo et al., 2016). We, therefore, gathered the total lines of code (LOC) and the number of classes for each version.

The collected data is analyzed by comparing the number of detected architectural smells in each project with Propagation Cost, Decoupling Level, and code coverage to identify correlations. The architectural smell data for each project is analyzed using the Shapiro–Wilks test, and it was found that it was not normally distributed. Therefore, correlations were measured using Spearman's correlation coefficient ρ. It is important to note that a negative statistical relationship with modularity/testability does not necessarily imply a negative ρ value. Specifically, a higher Propagation Cost indicates a decrease in modularity, while a higher Decoupling Level signifies an increase in modularity. For increased comprehensibility, the term "negative statistical relationship" is used interchangeably with "negative correlation", indicating that the involved quality attribute decreases as the other variable (e.g., smell or size) increases. Since the negative statistical relationship with Propagation Cost results in positive ρ values, the negative statistical relationships with Decoupling Level and code coverage are also presented as positive ρ values to facilitate analysis. As per convention, the p-value must be less than .05 to be statistically significant (Yin, 2018; Pigazzini et al., 2021a; Sas et al., 2022; Le et al., 2018), and $\rho \in [-0.1, 0.1]$ is treated as being too small to be significant. As potential confounding variables, correlations with LOC and the number of classes are compared to the detected smells and aforementioned metrics as well. Excluding Decoupling Level and Propagation Cost, which is only available for a project as a whole, this analysis is also performed on the package level by analyzing all unique packages in each project. In addition, a comparative analysis is conducted between different types of architectural smells. To assess potential differences, the correlations of architectural smells associated with impaired modularity or testability are compared to those of the non-impairing groups of smells. To analyze the findings, resulting empirically observable patterns are compared with this study's propositions, and the findings from each case are compared and contrasted in a cross-case synthesis.

3.4. Artifact

The principal artifact of this study, the Architectural Smell Analysis Tool (ASAT), served as the primary means of data collection and analysis. The code can be accessed[8] and a reproduction package is available.[9] ASAT consists of three distinct modules (see Fig. 2):

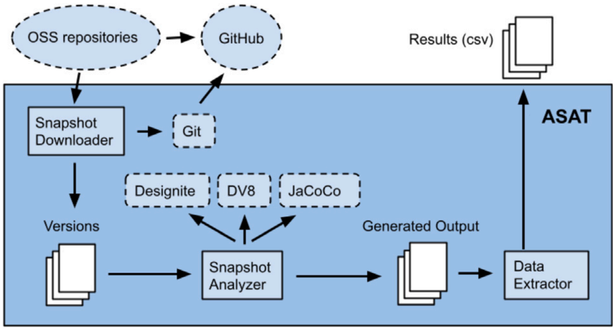

Fig. 2. Overview of the design of ASAT.

Snapshot Downloader is responsible for methodically retrieving versions at 4-week intervals from the inception to the most recent commit of the chosen GitHub repositories.

Snapshot Analyzer integrates three tools: Designite, JaCoCo, and DV8. Designite is utilized within ASAT to detect and calculate architectural smells within each version. JaCoCo measures testability in the form of code coverage. Finally, DV8 measures the modularity by calculating the Decoupling Level and Propagation Cost.

Data Extractor extracts and compiles the relevant data from the generated output for subsequent inspection and analysis.

4. Results

This section presents the study's findings, which are analyzed per the stated propositions. Additionally, the findings for the included cases are compared and contrasted to investigate patterns and tendencies.

RQ1a: Project-level modularity

Fig. 3 shows the correlations between architectural smells and the modularity metrics Propagation Cost and Decoupling Level. Mostly, there is no negative statistical relationship between modularity and architectural smells. Three out of eight cases (light-4j, yavi, and zip4j) showed a mostly negative statistical relationship. The Dense Structure smell, though, showed a negative statistical relationship in all cases where it was detected except for the seata project.

Most projects (i.e., cases) that we studied did not exhibit negative *correlations (i.e., statistical relationships)* between architectural smells and modularity at the project level, with the Dense Structure smell as an exception.

As a rival explanation, proposition D proposes that the negative statistical relationship between architectural smells and modularity can be explained by project size, whereby the decrease in modularity and the increase in smell both more strongly correlate with project size. Comparing modularity-smell correlations and modularity-size correlations (both LOC and number of classes) revealed that four out of eight cases (Myperf4J, seata, Sentinel, and sofa-rpc) clearly show a stronger negative statistical relationship between architectural smells and modularity than modularity and size. Interestingly, these four cases all rejected proposition A. In other words, the architectural smells in these projects showed a positive correlation with modularity, but the

[8] ASAT: https://doi.org/10.5281/zenodo.14853244

[9] Reproduction Package: https://doi.org/10.5281/zenodo.14853261

positive correlation between project size and modularity is stronger. In the remaining results, only a single case (yavi) accepts proposition D for modularity and project size, while the remaining three cases (light-4j, motan, and zip4j) show mixed results.

In contrast, comparing smell-modularity correlations and smell-size correlations shows that, in all cases but one, architectural smells generally have a stronger positive correlation with project size than a negative correlation with modularity. The remaining case, zip4j, showed mixed results. However, the Dense Structure smell showed a stronger correlation to project size in three out of the seven cases where it was detected.

In most cases, project-level modularity does not show a stronger negative *correlation (i.e., statistical relationship)* with project size than with architectural smells. This is largely because, even when there is a positive correlation between architectural smells and modularity, the positive correlation between project size and modularity is stronger. Additionally, architectural smells generally show a stronger correlation with project size than with modularity, except for the Dense Structure smell.

RQ1b: Project-level testability

Architectural smells and code coverage are predominantly negative (see Fig. 4). Two cases, sofa-rpc and zip4j, show a positive correlation instead. These two cases are the only ones that show a trend of increasing code coverage as the projects expanded. Another outlier is the Dense Structure smell that shows a negative statistical relationship with testability in one case—light-4j. All other cases show a positive correlation or are statistically insignificant ($p > .05$).

At the project level, six out of eight projects or cases have stronger negative correlations (i.e., statistical relationships) between testability and architectural smells. However, the same result is seen only for the Dense Structure smell in a single case.

As a rival explanation, the strength of the correlation between testability and architectural smells is compared to both the correlations between testability and project size, and architectural smells and project size. The results show that testability has stronger negative statistical relationships with size than with architectural smells for all but two cases (sofa-rpc and zip4j). The results mirror the project-level modularity results (RQ1a), where the cases that show a positive correlation with architectural smells also show an even stronger positive correlation to project size. In other words, the two cases where code coverage increased as the projects grew show more of a positive correlation with project size than smells.

By comparing the strength of negative correlations between architectural smells and testability and positive correlations between architectural smells and project size, we find that all cases show a stronger correlation to project size than to testability. Only the Dense Structure smell shows a stronger correlation with testability and only for the MyPerf4J project.

Overall, testability has a stronger negative *correlation (i.e., statistical relationship)* with project size than with architectural smells at the project level. However, the opposite is seen in cases with a positive correlation between testability and architectural smells. For architectural smells, all cases unanimously show a stronger correlation between architectural smells and project size compared to that between architectural smells and testability.

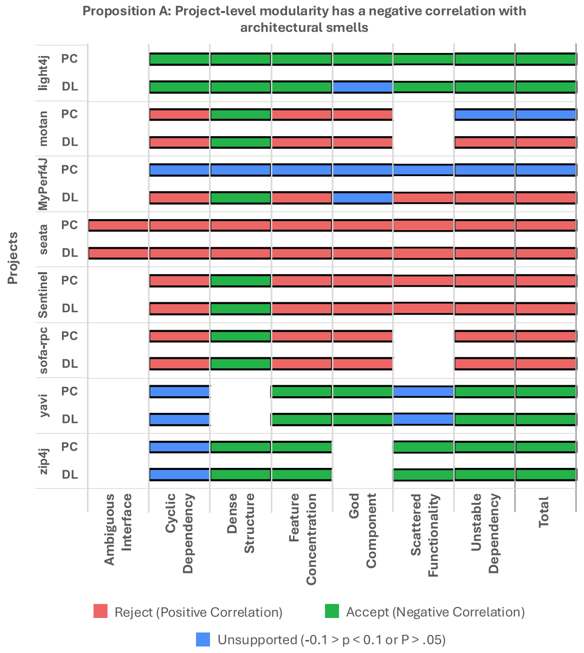

Fig. 3. Correlations (i.e., statistical relationships) between smells and the metric Propagation Cost (PC) and Decoupling Level (DL) at the project-level. This figure shows negative statistical relationships (Accept); positive statistical relationships (Reject); and statistically insignificant relationships, where either $\rho \in [-0.1, 0, 1]$ or $p > .05$.

RQ1c: Testability in packages

As shown in Fig. 5, most package-level detected architectural smells do not show a correlation with the code coverage in those packages.[10] Out of the eight cases, six predominantly do not establish correlations for most smells. Out of the remaining two cases, motan shows mostly positive correlation between three out of five categories of smells (God Component, Unstable Dependency, and total smells), thereby rejecting proposition B. The final case, MyPerf4J, shows a tie between smells with mainly established and mostly unsupported correlations, with two out of three primarily established correlations (Feature Concentration and Scattered Functionality) showing mainly negative statistical relationships.

Putting unsupported correlations aside, the two cases where code coverage increased with project size (sofa-rpc and zip4j) show the most positive correlations. However, unlike at the project level, the opposite result is not observed in the remaining cases. In total, five of the eight cases have mostly positive correlations. In contrast, only two cases show most negative statistical relationships (light-4j and MyPerf4J), and yavi is a tie. It is worth noting, however, that the cases zip4j and yavi present a relatively small number of unique packages, with only 17 and 18, respectively. Therefore, any conclusions drawn from those results should be treated with caution. The number of unique packages in each case can be seen in Table 1.

In most cases, correlations between architectural smells and package testability are statistically insignificant.

[10] No correlation means that either $p > .05$ or $\rho \in [-0.1, 0.1]$. For projects of constant smell measures throughout all versions, no ρ value could be calculated, and they are therefore excluded.

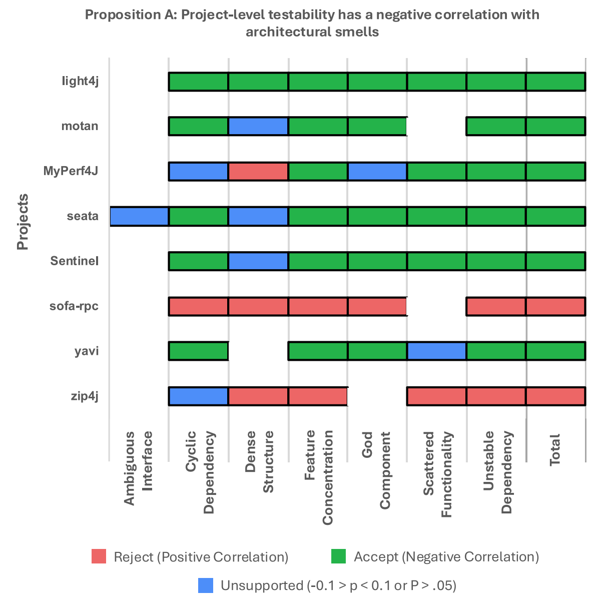

Fig. 4. Correlations (i.e., statistical relationships) between smells and the code coverage (at the project-level). This figure shows negative statistical relationships (Accept); positive statistical relationships (Reject); and statistically insignificant relationships, where $\rho \in [-0.1, .1]$ or $p > .05$.

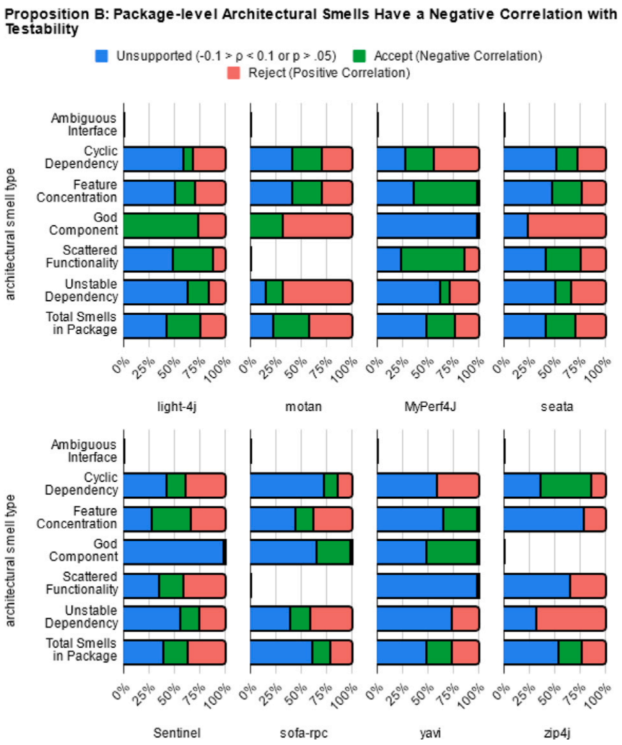

Fig. 5. Correlations (i.e., statistical relationships) between smells and the code coverage (package-level). Shows % of packages with negative statistical relationships (Accept); positive statistical relationships (Reject); and statistically insignificant relationships, where either $\rho \in [-0.1, 0.1]$ or $p > .05$.

As a rival explanation, the negative statistical relationships between testability and architectural smells in packages are compared to the negative statistical relationships between testability and project size and the positive statistical relationships between architectural smells and project size. The results show that the negative statistical relationships to testability and project size are stronger than to testability and smells among all classes. The same is true for all types of smells. Still, the statistical relationship between Cyclic Dependency and testability compared most favorably to project size, beating it in three cases and tieing in one.

In addition, the positive statistical relationship between architectural smells and project size is clearly stronger than the negative statistical relationship between architectural smells and testability in all cases. Only two outliers among the detected smells exist: Feature Concentration in yavi and Cyclic Dependency in zip4j.

All cases show that, within packages, both testability and architectural smells correlated more strongly with size than with each other.

RQ2a: Smell types and project-level modularity

This RQ aims to test whether architectural smells that are said to specifically impair modularity exhibit a stronger negative statistical relationship with it than other smells (proposition C). The results are shown in Table 2, where a higher ρ signifies a stronger negative statistical relationship with the associated metric Propagation Cost or Decoupling Level. Comparing the group mean of the smells in the impairing group (Ambiguous Interface, Cyclic Dependency, Dense Structure, God Component, and Scattered Functionality) to the smells in the non-impairing group (Feature Concentration and Unstable Dependency) shows that the impairing group has a stronger correlation with modularity in six out of eight cases.

The remaining two cases show mixed results, with one (light-4j) having a lower correlation with Propagation Cost in the impairing group and the other (zip4j) having a lower correlation with Decoupling Level in the impairing group. Interestingly, in most cases, two of the smells in the impairing group (God Component and Scattered Functionality) exhibit a lower correlation with modularity compared to the non-impairing smells. However, the rest of the impairing smells make up for the difference. Especially Dense Structure, which showed an average 98% higher correlation with modularity relative to the non-impairing smells.[11] Cyclic Dependency shows a stronger correlation in all cases except for two, seata and sofa-rpc, where it is slightly lower. Ambiguous Interface is only present in seata, but shows a 10% and 23% percent higher correlation with Propagation Cost and Decoupling Level respectively, relative to the group of non-impairing smells.

At the project level, three out of the five hypothesized modularity-impairing architectural smells show stronger negative statistical relationships compared to their non-modularity-impairing counterparts. These three smells are Ambiguous Interface, Cyclic Dependency, and Dense Structure. Especially, Dense Structure shows a strong negative statistical relationship with modularity. The two remaining hypothesized modularity-impairing smells, God Component and Scattered Functionality, show weaker negative statistical relationships compared to the non-impairing group of smells in most cases.

[11] Calculated using the combined averages of Propagation Cost and Decoupling Level.

Table 2
Project-level comparison of the ρ difference of architectural smells that impair modularity and their counterparts.

	light-4j		motan		MyPerf4J		seata		Sentinel		sofa-rpc		yavi		zip4j	
Smell	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL
Ambiguous Interface							0.069	0.194								
Cyclic Dependency	0.022	0.026	0.184	0.078		0.032	−0.010	−0.016	0.064	0.064	−0.046	−0.069				
Dense Structure	0.030	−0.021	1.119	0.961		0.985	0.184	0.289	1.354	1.359	1.247	1.221			0.006	0.498
God Component	−0.209		−0.017	−0.141			−0.080	−0.074	0.041	0.042	−0.005	−0.015	0.056	0.069		
Scattered Functionality	0.069	0.025	−0.096			−0.004	−0.148	−0.073	−0.004	−0.002					0.530	0.069
Group mean	−0.022	0.010	0.298	0.299		0.338	0.003	0.064	0.364	0.365	0.399	0.379	0.056	0.069	0.268	−0.048

Table 3 shows the ρ difference of (a) correlations between the indicated metric and modularity-impairing smells, and (b) the mean of the correlations between the indicated metric and non-modularity-impairing smells. Red indicates a weaker negative *correlation (or statistical relationship)*. PC: Propagation Cost, DL: Decoupling Level.

Table 3
ρ difference of Architectural Smells that impair Testability and their counterparts (Project-level).

Smell	light-4j	motan	MyPerf4J	seata	Sentinel	sofa-rpc	textbfyavi	zip4j
Dense Structure	−0.003		−0.858			0.232		−0.061
God Component	−0.073	−0.011		0.168	0.105	0.051	0.249	
Scattered Functionality	0.013		−0.007	−0.020	0.028			0.412
Group mean	−0.021	−0.011	−0.433	0.074	0.067	0.142	0.249	0.176

Table 4 shows the ρ difference of (a) correlations between testability-impairing smells and code coverage, and (b) the mean of the correlations between non-testability-impairing smells and code coverage. Red indicates a weaker negative *correlation (or statistical relationship)*.

RQ2b: Smells types and project-level testability

As shown in Table 3, the group mean of architectural smells that are said to impair testability (Dense Structure, God Component, and Scattered Functionality) show a stronger negative statistical relationship than non-impairing smells (Ambiguous Interface, Cyclic Dependency, Feature Concentration, and Unstable Dependency) in five out of eight cases. The motan and light-4j projects still exhibit strong correlations in their impairing smells, only losing to the non-impairing smells by a 1% and 2% relative difference, respectively. It should be noted that for motan, only a single type of testability-impairing smell is present, namely God Component.

Looking at the individual smells, Dense Structure only shows a stronger correlation with testability than the non-impairing smells in one case out of the four where it is present. The other impairing smells, God Component and Scattered Functionality, beat their non-impairing counterparts in most cases.

At the project level, two out of the three architectural smells that are supposed or hypothesized to impair testability show stronger negative statistical relationships than their non-impairing counterparts. These two smells are God Component and Scattered Functionality. The third smell (Dense Structure) shows the opposite result in most cases.

RQ2c: Smell types and testability in packages

The group mean of testability-impairing smells shows a stronger negative statistical relationship with testability in five out of eight cases, thus accepting proposition C. The remaining cases that rejected proposition C are the two largest projects containing the most packages (seata and Sentinel) and the smallest project with the fewest packages (zip4j). Notably, there is no overlap between which cases rejected proposition C at the project level (RQ2b) and the package level (RQ2c). The results are displayed in Table 4.

As for individual smells,[12] God Component has a higher correlation with testability than non-impairing smells in four out of five cases where it is present. Similarly, Scattered Functionality shows a higher correlation in three out of five cases where it is present, which aligns with the architectural smells on the project level. However, there is no overlap between the cases which did not show a higher correlation for these smells.

Feature Concentration, which is one of the non-testability-impairing smells, is tied in the number of stronger and weaker negative statistical relationships when compared to the group of testability-impairing smells. Notably, considering all results for both modularity and testability at the project and package level, it is the only non-impairing smell that did not exhibit a distinctly weaker negative statistical relationship.

Within packages, both God Component and Scattered Functionality, which are supposed or hypothesized to impair testability, show stronger negative statistical relationships than their non-impairing counterparts in the majority of cases.

5. Discussion

Contrary to expectations and hypothesized relationships (e.g., Rachow and Riebisch (2022), architectural smells did not show a negative correlation (i.e., statistical relationship) with modularity in RQ1a. A theory to explain this is that the absolute modularity of a project tends to increase with project size. As an example, Propagation Cost has been shown to decrease as the number of classes increases because as new classes are added, the number of dependencies among classes must also increase proportionately for the value to not decrease. However, the Decoupling Level should not be sensitive to size in this way, yet it also showed a positive statistical relationship with size in most cases. While both architectural smells and modularity tend to increase as a project grows in size, that positive relationship is likely a spurious relationship that can be explained by the increasing size.

It is also important to note that a single smell, Dense Structure, stood out from the rest in RQ1a. It showed a negative statistical relationship with modularity in six out of seven cases where it was detected and a stronger statistical relationship with modularity than size in four out of seven cases. This is no surprise, given how Dense Structure is measured. Dense Structure can only be detected at the project level for all packages and is binary in that it is either detected or not. To statistically measure relationships with Dense Structure, we used the *degree* measure instead of the number of detected instances. The *degree* is the average amount of dependencies between packages in a project, similar to how *Propagation Cost* defines the average dependencies between classes. Both measure coupling but at different levels of granularity.

On the other hand, the results of RQ1b showed that there is a negative statistical relationship between architectural smells and the test code coverage at the project level. Furthermore, architectural smells showed a weaker statistical relationship with code coverage than size. As with modularity, this also suggests a spurious relationship between architectural smells and code coverage, although this relationship was mostly negative rather than positive since, unlike modularity, testability tended to decrease in the observed cases. The increase in code coverage in two cases warrants further investigation. It is plausible that an undisclosed variable beyond size might contribute to this pattern. This assumption gains credibility considering the inconsistent positive or negative statistical relationships observed between size and testability. However, uncovering such a variable would require additional investigation and research.

The results for code coverage at the package level in RQ1c were quite different. First of all, the majority of detected architectural smells in packages did not show a correlation with code coverage. A possible theory for this is that many packages only exist in a limited number of versions — and the detected smells do not vary significantly across

[12] Dense Structure is not included since it is only detected at the project-level, and not in individual packages.

Table 4
ρ difference of Architectural Smells that impair Testability and their counterparts (Package-level).

Smell	light-4j	motan	MyPerf4J	seata	Sentinel	sofa-rpc	yavi	zip4j
God Component	0.240	0.200		−0.441		0.657	0.829	
Scattered Functionality	0.601		0.343	0.123	−0.059			−0.338
Group mean	0.420	0.200	0.343	−0.159	−0.059	0.657	0.829	−0.338

Table 5 shows the ρ difference of (a) correlations between testability-impairing smells and code coverage, and (b) the mean of the correlations between non-testability-impairing smells and code coverage. Red indicates a weaker negative *correlation (or statistical relationship)*.

those versions — making it difficult to establish a correlation. A similar pattern can also be observed in smaller packages, which tended to exhibit fewer smells. Even though they may exist in many versions, the smaller variance in smells may make it harder to establish statistically significant correlations. Looking at only established correlations in RQ1c, the results were the opposite of the project-level results, showing mostly positive correlations. Further, the negative statistical relationship with code coverage was found to be stronger with size than with smells. In other words, there does not seem to be a negative statistical relationship between architectural smells as a whole and code coverage in packages. However, looking at individual smells, Feature Concentration, God Component, and Scattered Functionality had more negative statistical relationships than positive ones. This shows that there is a significant difference between types of smells, and this finding is reflected in RQ2c as well.

While the results in RQ1 do not indicate that architectural smells are a causal factor, they do not preclude a common confounding factor. We have shown that size was a stronger factor than smells in both the negative statistical relationship with testability and the positive correlation with modularity. However, there may be another variable than size that is affecting both architectural smells and modularity/testability, such as technical debt. Several previous studies treat architectural smells as an indicator or symptom of technical debt (or a derivative thereof). But this would need to be investigated further. However, several tools provide computations of technical debt indices, accurately quantifying technical debt remains challenging due to the complex nature of software systems and the subjective aspects involved in evaluating code quality and maintainability (Kruchten et al., 2012).

Another theory, presented by Gnoyke et al. is that architectural smells are simply a result of a project's size rather than any architectural properties (Gnoyke et al., 2021). While this seems to align with the findings presented here, since overall, smells correlated more strongly with size than any other metric, this, too, would require further research to prove definitively. Another possible confounding factor is age, but it is difficult to isolate it from size since projects tend to grow over time. Therefore, it was not considered in this study; however, it is a factor worthy of future investigation.

Based on violated design principles and mentions in previous literature, it is proposed that Ambiguous Interface, Cyclic Dependency, Dense Structure, God Component, and Scattered Functionality should impair modularity and that Dense Structure, God Component, and Scattered Functionality should impair testability (Rachow and Riebisch, 2022). As shown in the results of RQ2, this was mostly the case. The smells said to impair modularity or testability displayed a higher negative statistical relationship for seven of the ten smells investigated at both package and project levels. Dense structure had an especially high affinity with modularity, which was a possible reason for this, as discussed earlier. However, Dense Structure did not show the same result with regard to testability. God Component and Scattered Functionality, on the other hand, did not show a stronger negative statistical relationship with modularity but for testability at both the project and package levels. Although not said to impair testability, Feature Concentration demonstrated a tie in the number of cases where it showed a stronger or weaker negative statistical relationship with testability in packages when compared to the group of impairing smells. But overall, no non-impairing smells showed a pattern of stronger negative statistical relationships than the group of affecting smells.

Table 5 shows the projects where the size indicated a greater correlation (i.e., statistical relationship) than the smells (and vice-versa) with the studied maintainability attributes; modularity and testability. The size of project *light-4j* shows a greater correlation than the smells with the decoupling level modularity metric. The size of projects *motan* and *zip4j* shows a greater correlation than the smells with the decoupling level modularity metric. The size of project *yavi* shows a greater correlation than the smells with the propagation cost and decoupling level metrics of modularity. The projects where the size shows a greater correlation than the smells with project-level testability are *light4j*, *motan*, *MyPerf4J*, *seata*, *Sentinel*, and *yavi*. On the other hand, the size of all the projects shows a greater correlation than the smells with package-level testability.

Table 6 shows the smells that negatively influence modularity and testability according to the findings of our study. Notably, the smells that we find to influence modularity negatively are *ambiguous interface*, *cyclic dependency*, and *dense structure*. The smells that negatively influence testability (both at project- and package-level) are *god component* and *scattered functionality*.

As far as we know, this is the first empirical result showing that different types of architectural smells relate to different sub-characteristics of maintainability. With this in mind, the results from RQ1c can be reexamined using only the smells that are said to impair testability — which now produces the opposite result with a majority of negative statistical relationships (if discounting statistically insignificant results). Previous research did not distinguish between the type of smell (Le et al., 2018) or did not see any significant difference in impact on the number of source code changes based on the kind of smell (Sas et al., 2022). The results support the notion that different smells can be used as indicators of potential quality issues. For example, God Component's and Scattered Functionality's smells are stronger indicators of reduced testability than their counterparts. This knowledge may help give developers clues about which parts of their systems have specific issues or where to prioritize refactorings.

Still, it is important to acknowledge that not every instance of an architectural smell is necessarily harmful but requires individual analysis. Some smells might be implemented on purpose as part of a solution, such as in certain design patterns. For instance, Martini et al. (2018) identified an instance of Unstable Dependency (in a project investigated in their study) that was caused by classes that applied a *Strategy* design pattern.

5.1. Lesson learned

We derive the following lessons from the study:

- *Size Matters*: The study found that the size of a project significantly impacts both modularity and testability. Larger projects tend to have higher modularity and lower testability, suggesting that size is a critical factor influencing these attributes.
- *Architectural Smells and Modularity*: Contrary to expectations, architectural smells did not show a negative statistical relationship with modularity. This indicates that modularity and architectural smells increase as projects grow, but this relationship might be spurious and driven by size.

Table 5
Projects where the project's size (or smell) showed a greater correlation (i.e., statistical relationship) with (a) project-level modularity (considering Decoupling Level (DL) and Propagation Cost (PC) metrics), (b) project-level testability, and (c) package-level testability. Green cells marked with 'Smell' mean that *smells* show a greater correlation than the size, red cells marked with 'Size' mean that *size* show a greater correlation than the smell, and blue cells marked with 'ND' means that there is No Difference.

light4j		**motan**		**MyPerf4J**		**seata**		**Sentinel**		**sofa-rpc**		**yavi**		**zip4j**	
DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC	DL	PC
Size	Smell	Size	ND	Smell	ND	Smell	Smell	Smell	Smell	Smell	Smell	Size	Size	Size	Smell

(a) Project-Level Modularity

light4j	**motan**	**MyPerf4J**	**seata**	**Sentinel**	**sofa-rpc**	**yavi**	**zip4j**
Size	Size	Size	Size	Size	Smell	Size	Smell

(b) Project-Level Testability

light4j	**motan**	**MyPerf4J**	**seata**	**Sentinel**	**sofa-rpc**	**yavi**	**zip4j**
Size	Size	Size	Size	Size	Size	Size	Size

(c) Package-Level Testability

Table 6
Smells that negatively influence modularity and testability, where (X) represents a negative influence.

Architectural Smell	Modularity	Testability	Package Testability
Ambiguous Interface	X		
Cyclic Dependency	X		
Dense Structure	X		
Feature Concentration			
God Component		X	X
Scattered Functionality		X	X
Unstable Dependency			

- *Dense Structure*: This particular architectural smell showed a consistent negative statistical relationship with modularity, highlighting its potential impact on the modularity of a project.
- *Testability and Code Coverage*: Architectural smells generally showed a negative statistical relationship with testability at the project level, but this relationship was weaker than the influence of size.
- *Package-Level Insights*: At the package level, most architectural smells did not correlate with code coverage, indicating that the impact of smells might be more pronounced at the project level than at the package level.

5.2. Implications

Since size significantly impacts modularity and testability, managing the growth of a project is crucial. Strategies to control size, such as modular design and refactoring, can help maintain quality. Moreover, educating developers about the impact of different architectural smells and the importance of managing project size can lead to better software design practices and improved maintainability.

Identifying and addressing specific architectural smells like Dense Structure, God Component, and Scattered Functionality can improve modularity and testability. Developers should prioritize refactoring efforts on these smells.

The study suggests that other variables, such as project age, might influence the relationship between architectural smells, modularity, and testability. Further research is needed to identify and quantify these factors.

The findings open avenues for further research, particularly in exploring the nuanced impacts of architectural smells at different project scales (e.g., package-level versus project-level) and concerning project size. Understanding these relationships in depth could lead to more effective strategies for managing architectural smells and maintaining software quality.

5.3. Threats to validity

This section discusses construct, external, reliability, and internal validity threats.

Construct validity concerns the suitability and accuracy of measurements used in the study. One such threat involves the choice of interval between versions. We here relied on a choice from previous research leading to reasonable extraction times through ASAT. Two included projects showed lower activity than desired, with, at times, more than four weeks between commits. This was a side-effect of prioritizing projects with uniform test coverage data. In addition, a few versions had to be discarded due to build issues, making it impossible to gather code coverage data.

Another possible threat is the uniformity of successful tests across versions since it might affect code coverage and, therefore, the reliability of the results. As shown in Fig. 1, the largest variance is in light-4j by 8.56%. However, this was a gradual decrease over six and a half years, where the number of tests went from less than 100 to over 800. Therefore, there was not a large fluctuation between versions, but code coverage data was not perfect because of failing tests. Only the yavi project had a 100% test success rate across all versions.

We used code coverage to measure testability. Of course, code coverage can be influenced by outside factors such as the development culture in a project. In addition, one aspect of testability — the "efficiency with which test criteria can be established" — is hard to measure. Code coverage does not account for the developer effort required for test implementation. Greater confidence in the findings is advised to be obtained through interviews or surveys with developers and architects to evaluate whether the negative relationship with test writing efficiency can be confirmed.

We acknowledge that the validity of our results may be influenced by the choice of tools used for metrics computation and smell detection. Different tools often implement varying algorithms, heuristics, and detection rules, which can lead to discrepancies in the results. For instance, detecting code smells can be particularly sensitive to the specific rules and thresholds employed by the tools. These variations can result in different tools identifying different sets of smells or even interpreting the severity of the same smells differently.

To mitigate this threat, we have carefully selected widely recognized and validated tools in the research community and added the correct metric formulations in our method section. However, we acknowledge that these tools have limitations, and results might differ if alternative tools were used. We encourage further research and replication of our study using different tools to better understand the robustness of our findings.

External validity is concerned with the generalizability of the study's findings. While all projects are open-source and implemented in Java, we selected them from diverse domains and of varying sizes. Moreover, only non-trivial projects with sufficient activity were included. However, none of the included projects are older than seven years or have more than 250k LOC. Also, while eight cases are not low compared to many other recent studies about architectural smells (see, e.g., Le et al. (2018), a larger dataset would justify a greater generalizability of the findings.

Another threat concerns the fact that this study is not exhaustive in the architectural smells, i.e., other ones are not included here. Nothing can be said about them. Also, the Ambiguous Interface smell was only present in a single case.

Reliability is concerned with the replicability of the study. This study follows a well-established case study design framework by Yin (2018) with transparency regarding the methodology used to produce its results. ASAT and all collected data are freely available online for study or replication (see links above). It is worth noting that the integrated tools used to perform static analysis on architectural smells (Designite) and modularity metrics (DV8) require an academic or paid license.

Internal validity threats do not apply as we do not try to establish any causal relationships.

6. Conclusion

We make three main observations. First, there was no negative statistical relationship between project-level modularity and architectural smells as a whole in most cases. However, the Dense Structure smell stood out as an exception, showing a strong negative statistical relationship with modularity—even stronger than its statistical relationship with project size. Second, a negative statistical relationship was observed between architectural smells and project-level testability. However, project size shows a stronger statistical relationship with both, suggesting it may be a spurious association. Third, the analysis of package-level smells revealed that they generally did not exhibit significant statistical relationships with testability, and when statistical relationships were present, they were not predominantly negative. Fourthly, seven out of the ten examined architectural smells demonstrated a stronger negative statistical relationship with the specific quality attributes they were claimed to impair compared to their counterparts. Three out of five smells associated with modularity impairment exhibited higher negative v relationships at the project level. These smells are Ambiguous Interface, Cyclic Dependency, and Dense Structure. For testability, the associated smells God Component and Scattered Functionality displayed higher negative statistical relationships at both the project and package levels.

For software developers, the study offers insights into the significance of specific architectural smells in their own projects. For researchers, further empirical investigations of the impact of architectural smells on software maintainability are necessary to gain a deeper understanding of the impact and develop more effective strategies for managing architectural smells and maintaining software quality.

6.1. Future work

Exploring ways to quantify additional sub-characteristics of maintainability could reveal more relationships with different architectural smells. In particular, studies can consider analyzability, modifiability, and reusability to investigate whether these quality attributes also have a stronger negative statistical relationship with the smells that are reportedly said to impair them. Considering analyzability, we propose the use of metrics such as cyclomatic complexity and code churn. Cyclomatic complexity measures the complexity of the code, which can impact how easily it can be analyzed (Ebert et al., 2016). Whereas code churn tracks the frequency of changes in the codebase, indicating potential areas that are harder to analyze (Hall and Munson, 2000). To investigate modifiability and reusability, we propose the use of metrics such as Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), component reusability, and inheritance depth (Chidamber and Kemerer, 1994).

Moreover, we have theorized that the observed increase in architectural smells and modularity over time might be a natural consequence of growing projects. However, other confounding factors may be responsible for the observed correlations (i.e., statistical relationships) between architectural smells and modularity/testability, including, e.g., project age. Hence, additional research is required to identify and measure these confounding factors.

CRediT authorship contribution statement

Rodi Jolak: Writing – review & editing, Supervision, Methodology, Investigation. **Simon Karlsson:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Felix Dobslaw:** Writing – review & editing, Supervision, Project administration.

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

The artifacts and data are provided in the manuscript as footnotes in Section 3.4 Artifact.

References

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. http://dx.doi.org/10.1016/j.jss.2019.04.066.

Azadi, U., Arcelli Fontana, F., Taibi, D., 2019. Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 88–97. http://dx.doi.org/10.1109/TechDebt.2019.00027.

Baabad, A., Zulzalil, H.B., Hassan, S., Baharom, S.B., 2020. Software architecture degradation in open source software: A systematic literature review. IEEE Access 8, 173681–173709. http://dx.doi.org/10.1109/ACCESS.2020.3024671.

Capilla, R., Fontana, F.A., Mikkonen, T., Bacchiega, P., Salamanca, V., 2023. Detecting architecture debt in micro-service open-source projects. In: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 394–401.

Chantian, B., Muenchaisri, P., 2019. A refactoring approach for too large packages using community detection and dependency-based impacts. In: Proceedings of the 1st World Symposium on Software Engineering. pp. 27–31. http://dx.doi.org/10.1145/3362125.3362132.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Cunningham, W., 1992. The WyCash portfolio management system. SIGPLAN OOPS Mess. 4 (2), 29–30. http://dx.doi.org/10.1145/157710.157715.

De Andrade, H.S., Almeida, E., Crnkovic, I., 2014. Architectural bad smells in software product lines: An exploratory study. In: Proceedings of the WICSA 2014 Companion Volume. pp. 1–6.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P., 2016. Cyclomatic complexity. IEEE Softw. 33 (6), 27–29.

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

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009a. Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering. IEEE, pp. 255–258. http://dx.doi.org/10.1109/CSMR.2009.59.

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009b. Toward a catalogue of architectural bad smells. In: Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5. Springer, pp. 146–162.

Gnoyke, P., Schulze, S., Krüger, J., 2021. An evolutionary analysis of software-architecture smells. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 413–424. http://dx.doi.org/10.1109/ICSME52107.2021.00043.

Hall, G.A., Munson, J.C., 2000. Software evolution: code delta and code churn. J. Syst. Softw. 54 (2), 111–118.

ISO/IEC 25010:2023, 2023. International Standard for Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation. 2023, Standard. International Organization for Standardization, Geneva.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. IEEE Softw. 29 (6), 18–21. http://dx.doi.org/10.1109/MS.2012.167.

Le, D.M., Carrillo, C., Capilla, R., Medvidovic, N., 2016. Relating architectural decay and sustainability of software systems. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture. WICSA, IEEE, pp. 178–181.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018. An empirical study of architectural decay in open-source software. In: 2018 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 176–17609. http://dx.doi.org/10.1109/ICSA.2018.00027.

Lee Rodgers, J., Nicewander, W.A., 1988. Thirteen ways to look at the correlation coefficient. Amer. Statist. 42 (1), 59–66.

Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley.

MacCormack, A., Rusnak, J., Baldwin, C.Y., 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Manag. Sci. 52 (7), 1015–1030. http://dx.doi.org/10.1287/mnsc.1060.0552.

Martin, R.C., 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR.

Martini, A., Fontana, F.A., Biaggi, A., Roveda, R., 2018. Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In: Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12. Springer, pp. 320–335. http://dx.doi.org/10.1007/978-3-030-00761-4_21.

Mehboob, B., Chong, C.Y., Lee, S.P., Lim, J.M.Y., 2021. Reusability affecting factors and software metrics for reusability: A systematic literature review. Software: Pr. Exp. 51 (6), 1416–1458. http://dx.doi.org/10.1002/spe.2961.

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

Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q., 2016. Decoupling level: A new metric for architectural maintenance complexity. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, pp. 499–510. http://dx.doi.org/10.1145/2884781.2884825.

Mo, R., Snipes, W., Cai, Y., Ramaswamy, S., Kazman, R., Naedele, M., 2018. Experiences applying automated architecture analysis tool suites. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 779–789. http://dx.doi.org/10.1145/3238147.3240467.

Mumtaz, H., Singh, P., Blincoe, K., 2021. A systematic mapping study on architectural smells detection. J. Syst. Softw. 173, 110885. http://dx.doi.org/10.1016/j.jss.2020.110885.

Pigazzini, I., Fontana, F.A., Walter, B., 2021a. A study on correlations between architectural smells and design patterns. J. Syst. Softw. 178, 110984. http://dx.doi.org/10.1016/j.jss.2021.110984.

Pigazzini, I., Fontana, F.A., Walter, B., 2021b. A study on correlations between architectural smells and design patterns. J. Syst. Softw. 178, 110984. http://dx.doi.org/10.1016/j.jss.2021.110984.

Rachow, P., Riebisch, M., 2022. An architecture smell knowledge base for managing architecture technical debt. In: Proceedings of the International Conference on Technical Debt. pp. 1–10. http://dx.doi.org/10.1145/3524843.3528092.

Sas, D., Avgeriou, P., Kruizinga, R., Scheedler, R., 2021. Exploring the relation between co-changes and architectural smells. SN Comput. Sci. 2, 1–15. http://dx.doi.org/10.1007/s42979-020-00407-5.

Sas, D., Avgeriou, P., Pigazzini, I., Arcelli Fontana, F., 2022. On the relation between architectural smells and source code changes. J. Software: Evol. Process. 34 (1), e2398. http://dx.doi.org/10.1002/smr.2398.

Sharma, T., Fragkoulis, M., Spinellis, D., 2016. Does your configuration code smell? In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 189–200.

Sharma, T., Kessentini, M., 2021. QScored: A large dataset of code smells and quality metrics. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, pp. 590–594. http://dx.doi.org/10.1109/MSR52588.2021.00080.

Sharma, T., Singh, P., Spinellis, D., 2020. An empirical investigation on the relationship between design and architecture smells. Empir. Softw. Eng. 25, 4020–4068.

Stochel, M.G., Chołda, P., Wawrowski, M.R., 2020. On coherence in technical debt research : Awareness of the risks stemming from the metaphorical origin and relevant remediation strategies. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, pp. 367–375. http://dx.doi.org/10.1109/SEAA51224.2020.00067.

Tian, F., Liang, P., Babar, M.A., 2019. How developers discuss architecture smells? an exploratory study on stack overflow. In: 2019 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 91–100. http://dx.doi.org/10.1109/ICSA.2019.00018.

Verdecchia, R., Kruchten, P., Lago, P., 2020. Architectural technical debt: A grounded theory. In: Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14. Springer, pp. 202–219.

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

Yasi, T., Qin, J., 2022. Automatic building of java projects on GitHub: A study on reproducibility. URL https://www.diva-portal.org/smash/get/diva2:1691841/FULLTEXT01.pdf.

Yin, R.K., 2018. Case Study Research and Applications, sixth ed. SAGE Publications, Thousand Oaks, CA.