Mittuniversitetet
MID SWEDEN UNIVERSITY

Studying the Relationship between Architectural Smells and Maintainability

Alexander Berglund and Simon Karlsson

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/year: Spring 2023
Supervisor: Rodi Jolak
Examiner: Felix Dobslaw
Course code/registration number: DT133G
Programme: Software Engineering

Studying the Relationship between Architectural Smells and Maintainability

Simon Karlsson and Alexander Berglund
Department of Communication, Quality Management and Information Systems
Mid Sweden University
Östersund, Sweden
{sika2001, albe2003}@student.miun.se

***Abstract*—In recent years, there has been a surge in research on the impact of architectural smells on software maintainability. Maintainability in turn encompasses several other quality attributes as sub-characteristics, such as modularity and testability. However, the empirical evidence establishing a clear relationship between these quality attributes and architectural smells has been lacking. This study aims to fill this gap by examining the correlation between seven architectural smells and testability/modularity across 378 versions of eight open-source projects. A self-developed tool—ASAT—was used to collect data on architectural smells and metrics relating to modularity and testability. The collected data was analyzed to reveal correlations at both the project-level and within packages. Contrary to expectations, the findings show that, generally, there is no negative correlation between smells and modularity at the project-level, except for the Dense Structure smell. Remarkably, project-level testability showed the opposite result. However, a rival explanation proposes that the increasing size of a project may be a stronger factor in this relationship. Similarly, package-level smells, as a whole, did not exhibit a negative correlation with testability. However, most smells demonstrated a stronger negative relationship with the quality attributes they were claimed to impair, in comparison to their counterparts. This empirical evidence substantiates the assertion that specific architectural smells indeed relate to distinct quality attributes, which had previously only been supported by argument.**

***Index Terms*—architectural smells, software architecture, software maintenance, software metrics, software quality**

Acknowledgement

We would like to thank Rodi Jolak and Felix Dobslaw. Without their patience, guidance, and support this study would not have been possible.

I. Introduction

Architectural smells are recurring, identifiable architectural design decisions that mainly impact the maintainability of a system [1]–[4]. 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 [5]. In a study comparing the two, Fontana et al. found that there was no correlation between code smells and architectural smells, and that addressing one did not affect the other [6]. They conclude that architectural smells are "independent from code smells, and therefore deserve special attention by researchers, who should investigate their actual harmfulness."

Recent studies show that architectural smells are prevalent in open-source projects. In a dataset of more than 86 thousand Java and C# GitHub repositories, Sharma et al. [7] found nearly 1.2 million instances of seven distinct types of architectural smells. Meanwhile, Fontana et al. [6] observed at least two different types of architectural smells in 102 out of 103 open-source projects they investigated. Notably, they only measured three unique types of architectural smells, indicating that the actual prevalence of architectural smells might have been even higher.

However, architectural smells lack a universal definition, leading to different interpretations among researchers. One widely-used definition [5] considers them a cause of quality degradation, while another [8] views them as indicators of potential problems that require individual analysis. Recent research reflects this lack of consensus, with some researchers treating architectural smells as symptoms of underlying issues such as technical debt or architectural decay [3], [9], [10], while others treat them as a measure of software quality itself [7], [11].

The assertion that architectural smells negatively impact quality has been criticized as being based on individual developer experience and intuition, rather than qualitative data [1], [9]. Some recent studies show that certain architectural smells can negatively impact the number of issues and changes in open-source projects [4], [9]. However, other recent studies show that sometimes architectural smells are benign, or even part of a deliberate solution [3], [12]. 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 [11].

Architectural smells have been gaining increased attention in recent years, with a surge in published articles, as well as new tools to detect smells. Many of these tools require a license and may claim to improve software quality and reduce costs [13]–[15]. Detected smells may impact the development of a project, such as where to spend development time and resources. However, refactoring an architectural smell is a nontrivial task, that may introduce other smells in the process [16]. Considering these factors, there is a need for quantitative data to measure the impact of individual smells to help developers make informed decisions.

The objective of this paper is to collect quantitative data on the impact of architectural smells on maintainability by examining various open-source software projects. The paper aims to investigate how detected architectural smells affect certain quality attributes, specifically modularity and testability, which are sub-characteristics of maintainability. By examining the relationship over time, the study aims to uncover the nature of this relationship. Hopefully, this will contribute to a more consistent definition of architectural smells. Furthermore, the study will analyze individual types of architectural smells to determine how they influence the measured quality attributes, providing more detailed insights into the effects of each smell.

II. Purpose and Contributions

Architectural smells are often claimed to negatively impact software quality, specifically maintainability, but there is a lack of quantitative data. Maintainability comprises several sub-characteristics[1], and individual architectural smells are inferred to impair certain sub-characteristics based on violated design principles [1], as shown in Table I. This, however, has not been empirically measured. In a systematic mapping study on architectural smells detection, Mumtaz et al. stress that it is paramount to investigate the sub-characteristics of a quality characteristic such as maintainability [18]. They suggest that future studies should explore methods for measuring and analyzing the relationship between architectural smells and these sub-characteristics—including testability in particular, which has received limited attention in research thus far.

TABLE I
Quality Attributes Impaired by Architectural Smells

Smell	Analyzability	Modularity	Reusability	Modifiability	Testability
Ambiguous Interface	X	X	X		
Cyclic Dependency	X	X	X	X	
Dense Structure	X	X	X	X	X
Feature Concentration	X			X	
God Component	X	X	X	X	X
Scattered Functionality	X	X	X	X	X
Unstable Dependency					

Shows the architectural smells considered in this study, and the sub-characteristics of maintainability that they impair according to Rachow et al. [1].

This project aims to gain a deeper understanding of the relationship between architectural smells and two of the quality attributes that they are said to impair: modularity and testability. To do so, the evolution of architectural smells and specific metrics related to modularity and testability will be investigated across several diverse open-source projects.

The research questions that will be answered are:

[1]The sub-characteristics of maintainability according to ISO/IEC 25010:2011 [17] are modularity, reusability, analyzability, modifiability, and testability.

RQ1: What is the relationship between architectural smells and:
RQ1a: Project-level modularity?
RQ1b: Project-level testability?
RQ1c: Testability in packages?
RQ2: How do negative relationships compare between different types of architectural smells and:
RQ2a: Project-level modularity?
RQ2b: Project-level testability?
RQ2c: Testability in packages?

The purpose of RQ1 is to establish whether there is a negative relationship between the measured architectural smells and quality attributes, where the quality attributes decrease as architectural smells increase, and vice versa. RQ2 looks at each measured type of architectural smell individually to see how they relate to the measured quality attributes. The purpose is to examine whether smells that are said to impair modularity or testability exhibit stronger negative relationships compared to their counterparts.

The contributions of this study are two-fold: first and foremost, this study provides empirical evidence highlighting a clear distinction among different types of architectural smells concerning their relationships with modularity and testability. By confirming previously held theoretical assumptions about architectural smells, this initial step validates existing knowledge and paves the way for future research, which can further explore the connections between specific architectural smells and quality attributes. For software developers, the results can help provide practical guidance regarding the architectural smells present in their own projects, by providing insights on the specific issues they may indicate.

Second, this study introduces a tool named ASAT that researchers can use to gather data about architectural smells, modularity, and/or testability. ASAT offers an efficient means of data collection by downloading and batch processing multiple versions of open-source projects from GitHub. Additionally, software developers can potentially benefit from this tool by obtaining concrete and quantitative metrics regarding their own projects over time.

III. Background

This section presents a brief overview of architectural smells, and the related concepts of software quality, maintainability, design principles, and technical debt. Each concept is described in the associated subsection.

A. Architectural Smells

Architectural smells are recurring solutions at the architectural level of a project that, while not erroneous, can still pose a problem [5]. They are mainly characterized as violating design principles [1], [8], [19], and negatively impacting the maintainability of a system [1]–[5]. Since architectural smells operate on the architectural level, they involve components such as classes, packages, and subsystems, and as such, require larger refactorings to remove [8].

An example of an architectural smell is the Cyclic Dependency smell. It is claimed to impair all sub-characteristics of maintainability and is therefore thought to be one of the most critical smells [1], [20]. It refers to two or more components that depend on each other, forming a cycle. A component can be either a class or a package—or even a whole subsystem [8]. The dependency "chain" doesn't have to be a circle, but can take many different shapes, as illustrated by Figure 1. Lippert & Roock explain that cyclic dependencies can affect maintainability negatively since they can have "severe and unpredictable consequences"—modifications to one part of the cycle can have side-effects in any other part of the cycle [8].

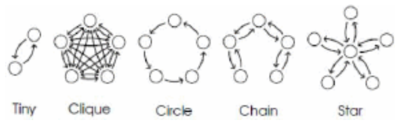


Fig. 1. Various shapes of Cyclic Dependencies. From Azadi et al. [19]

B. Software Quality

According to the ISO/IEC 25010:2011 standard [17], there are two distinct facets of software quality: 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 concerned with the latter. The standard's product quality model comprises eight characteristics (functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability) and their related sub-characteristics. The purpose of the model is to aid stakeholders in specifying and evaluating the quality of a software system. This can involve metrics that measure certain properties of the source code. However, it is sometimes difficult to quantify quality characteristics to provide an appropriate measurable value [21].

C. Maintainability

This study uses the concept of maintainability and its sub-characteristics, as defined by the ISO/IEC 25010:2011 standard [17]. 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. It can be viewed as either an inherent capability of the system to facilitate maintenance or the quality in use experienced by the maintainers. Given that software often has a long lifespan and evolves over time, maintainability is a critical aspect of software engineering. Maintainability consists of the sub-characteristics modularity, reusability, analyzability, modifiability, and testability. In this study, only modularity and testability are considered.

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.

D. Design Principles

According to Lippert & Rook [8], architectural smells occur when recognized design principles are violated. Therefore, adhering to these design principles can provide a starting point for developing a better system structure. Martin [22] describes the SOLID design principles in relation to software architecture as guidelines for arranging functions and data structures into classes and interconnecting those classes to create well-designed systems. Moreover, the principles can also be applied to larger architectural building blocks. In addition to SOLID, there are other well-established design principles such as the DRY—or Don't Repeat Yourself—principle which stipulates that you should not write the same or similar code more than once.

E. Technical Debt

In 1992, Cunningham introduced the use of debt as a metaphor in software development [23]. 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—what we now refer to as refactoring. Failing to address this debt can impede progress and hinder future development efforts.

In the ensuing decades, the metaphor has become known as technical debt and has grown in both adoption and meaning. Kruchten et al. describe it as "the invisible result of past decisions about software that negatively affect its future", with emphasis on the "invisible" aspect [24]. They assert that technical debt can impede a system's evolvability and maintainability. Furthermore, many different elements can contribute to technical debt, such as a system's architecture in the form of architecture debt. Martini et al. further expand on this concept, using the term Architecture Technical Debt to mean sub-optimal architecture decisions that violate the intended architecture [25].

The phenomenon of architecture deviating from its intended design has been defined in many ways. Baabad et al. identify several related concepts including architectural degeneration, erosion, drift, mismatch, decay, degradation, and more [10]. In their study on the use of technical debt terminology, Stochel et al. conclude that there is ambiguity in how the various terms are applied in research [26].

IV. Related Work

The first architectural smells were defined by Lippert & Roock [8], which included the Cyclic Dependency smell that has been cited as the most impactful smell [1], [20]. However, they note that this smell can be a part of certain design patterns and can be harmless in such cases. They argue that not all smells are problematic, and need to be analyzed on a case-by-case basis. Subsequently, Garcia et al. defined four new architectural smells, and characterized architectural smells as a commonly used architectural decision that negatively impacts system quality, whether intentional or not [5]. This notion that architectural smells always impact software negatively has become widespread. However, in a study on the relationship between design patterns and architectural smells, Pigazzini et al. found that some patterns can introduce smells [27], as noted by Lippert & Roock [8]. They label these smells as false positives since they are present because developers had intended to program them. Martini et al. observe a similar result in interviews and questionnaires with developers about their own systems, where many Cyclic Dependencies were part of a deliberate solution according to the developers themselves [3].

Presently, numerous other architectural smells have been identified. In a systematic mapping study, Mumtaz et al. [18] found and described 108 architectural smells. However, many of these smells are not detectable by existing tools or techniques. Azadi et al. [19] presented a catalog of 12 architectural smells that are detectable by current tools, classified according to violated design principles. Subsequently, Rachow et al. [1] 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. Based on the fact that design principles promote quality, they argued that a violation of 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[2]. However, the authors acknowledged that in general the connections between architectural smells and quality attributes or design principles were merely asserted or supported by argument, rather than being demonstrated empirically.

Gnoyke et al. [20] 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 remained mostly 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 are actually a problem.

Le et al. [9] 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.

Sas et al. [4] 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 have larger changes. They also found that as the number of smells that affect a component increase, so does the likelihood that it will change. The type of architectural smell did not have a significant correlation with changes, but they note that potentially not all types of cycles are impactful on changes, which is consistent with previous research [3], [8], [27]. However, they found that the introduction of a smell only increased the number of changes in the affected component in some cases—sometimes the opposite was true.

Similar to the aforementioned empirical studies, this study will also measure architecture smells in several versions of open-source projects over time. It will employ a methodology inspired by Sas et al. [4] 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 the best of 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 important subject for future research [18]. Furthermore, this will help verify the links between individual architectural smells and these sub-characteristics in the knowledge base by Rachow et al. [1], which currently lacks supporting empirical evidence.

V. Research Methodology

The study was conducted as an embedded multiple case study [28]. The case study design is presented in Figure 2. Each case consists of an open-source Java project with multiple units of analysis in a chronological sequence. Specifically, each unit of analysis is a version of the project at a given point in its development[3].

A. 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: There is a negative relationship between architectural smells as a whole and modularity and testability.** Architectural smells are often described

[2]Table I shows the sub-characteristics impaired by the architectural smells considered in this study according to the knowledge base.

[3]In other words, a commit to the project's main branch on GitHub.

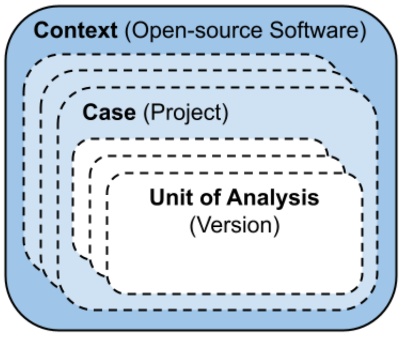


Fig. 2. Case study design.

as impacting maintainability negatively [1]–[4], which implies they should also affect testability and modularity since they are sub-characteristics of maintainability. Consequently, architectural smells should have a negative relationship with the modularity and testability of a system as a whole.

- **Proposition B: There is a negative relationship between architectural smells in individual packages and the testability of those packages.** Previous studies have found a correlation in components between architectural smells and changes and issues [4], [9], 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 relationship with architectural smells.
- **Proposition C: Types of architectural smells that are said to specifically impair modularity and/or testability exhibit a stronger negative relationship with those attributes.** Rachow et al. [1] presented a knowledge base of architectural smells and the sub-characteristics of maintainability that they are said to impair, based on violated design principles and stated links between architectural smells and quality attributes in research. Based on this, certain types of architectural smells should have a stronger negative relationship with associated sub-characteristics.

Furthermore, we present a contrasting proposition as a rival explanation.

- **Proposition D: There is a stronger relationship between architectural smells and size, as well as modularity/testability and size, than the relationship between smells and modularity/testability.** Gnoyke et al. theorized that architectural smells might be a symptom of the size of a project, rather than any architectural properties [20]. As a rival explanation, size may be a confounding factor that explains the observed correlations between architectural smells and modularity/testability, whereby size explains both the increase in architectural smells and the decrease in modularity/testability.

B. Cases

To ensure a representative sample of open-source projects across diverse domains, eight projects (see Table II) were selected from GitHub based on the following inclusion criteria, inspired by Sas et al. [4]:

- The project is nontrivial and has a size of at least 10,000 lines of code in its latest commit.
- The project contains at least one instance of an architectural smell in its latest commit.
- The project has been in active development for at least three years.
- The project shows consistent activity during development[4].

The projects were analyzed over their entire lifetime, providing a total of 378 versions with roughly 29 years of active development data.

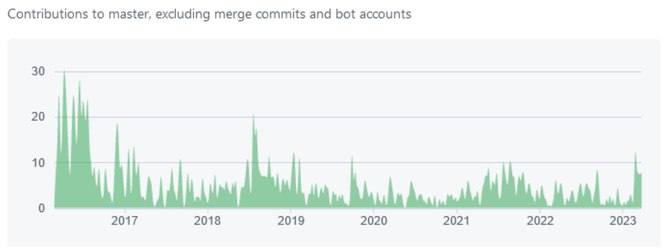
All the cases included in this study were free and open-source projects. Open-source projects provide a transparent and collaborative environment, where source code is freely accessible to the public. This accessibility eliminates any potential ownership issues, as the projects are typically governed by open licenses that grant the freedom to view, modify, and distribute the code. Moreover, the nature of open-source projects promotes a culture of shared knowledge and collaboration. Developers who contribute to these projects understand and expect that their code will be publicly available for scrutiny and analysis. Therefore, the gathering and analysis of source code from these projects pose no ethical concerns regarding ownership or privacy.

C. Data Collection and Analysis

For each project, versions were taken at 4-week intervals from project inception to the latest commit. This interval has been used in previous research, where it was found to produce a meaningful level of change in calculated metrics between versions [4]. Moreover, using a fixed interval for all cases ensures consistent results without biases. However, using a lower interval would provide more granular results, but would also require additional processing time due to increased computational demands.

For each version, Designite[5] was used to detect the number of each architectural smell Ambiguous Interface, Cyclic Dependency, Feature Concentration, God Component, Scattered

[4]As shown by the Contributors page on GitHub. Here is an example of what a project showing consistent, active development across its lifetime might look like:



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

TABLE II
PROJECTS SELECTED FOR ANALYSIS

Description	**Project Name**	**First Version**	**Versions**	**Unique Packages**	**LOC in Last Version**	**Classes in Last Version**	**Architectural Smells in Last Version**
Microservices framework	light-4j	2016-10-09	76	115	68k	906	101
RPC framework	motan	2016-04-20	61	76	43k	621	108
Performance monitoring tool	MyPerf4J	2018-11-04	33	118	17k	322	56
Distributed transaction solution	seata	2019-02-10	51	372	238k	2527	405
Flow control component	Sentinel	2018-08-24	50	366	181k	1516	354
RPC framework	sofa-rpc	2018-06-08	46	186	84k	1239	189
Validation framework	yavi	2018-08-21	33	18	20k	533	40
Java library	zip4j	2019-07-04	28	17	15k	150	56

Functionality, and Unstable Dependency. As for the Dense Structure smell, it is either present or absent in a project, leading to a binary value representation. Since a binary value wouldn't be conducive to performing a meaningful analysis, the average degree of each detected Dense Structure was used instead, which measures the intensity of the smell. This is discussed further in the Discussion section. In addition, metrics relating to modularity and testability were measured.

Modularity is defined as the degree to which a system is made up of discrete components that can be changed with minimal effect on each other [17]. The first property was measured as Decoupling Level [29], which measures how well a system is divided into discrete modules. The second property was measured by Propagation Cost [30], which measures how tightly coupled a system is. In a study of eight industry projects, Mo et al. found that the two measures reflected the knowledge of the software architecture by the architects themselves, and that they helped the architects present architecture quality issues in a quantitative way [31]. Propagation Cost and Decoupling Level were measured using the tool DV8[6].

Testability is defined as 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 [17]. This was measured by the code coverage—or the percentage of lines of code that are executed by tests. For each version, the code coverage was measured using JaCoCo[7]. If there is a reduction in the effectiveness and efficiency of establishing test criteria and the ability to perform tests, it is likely to negatively impact the code coverage.

Measuring architectural smells and the metrics related to modularity was done using static analysis of the source code. However, measuring the code coverage of each version required dynamic analysis. In other words, each version had to be built and have its unit tests run in order to generate the results. As reported by Yasi et al., not only are build failures exceedingly common, the leading root cause of build failures is test failure [32]. As a consequence, project selection was heavily influenced by the ability to build a project and successfully run its tests. Projects where code coverage results were deemed to be unreliable, due to either a high number of unsuccessful tests or sudden fluctuation in test success rate between versions were excluded. In addition, certain versions which would not build, such as the initial commit in several projects, were excluded from the analysis. Figure 3 displays boxplots of the test success rate across all included versions in the selected projects. This is also discussed in further detail in the Threats to Validity section.

[6]DV8 can be found here: https://archdia.com/.
[7]JaCoCo can be found here: https://www.jacoco.org/jacoco/trunk/index.html.

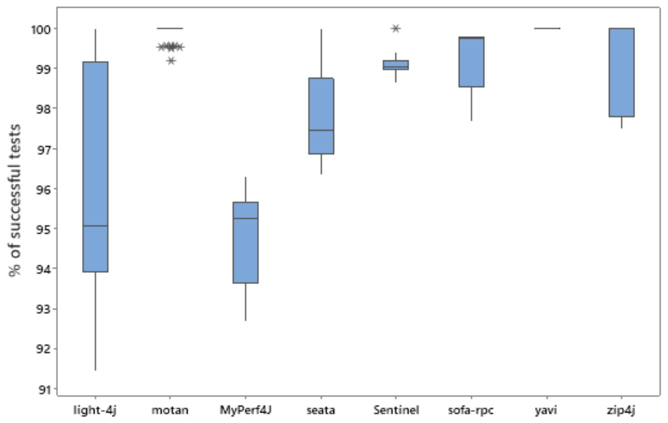


Fig. 3. The percentage of successful tests in all versions of the selected projects.

Size has been identified as a potential confounding variable. Gnoyke et al. theorized that the quantity of source code might affect the occurrence of architectural smells [20]. Furthermore, the number of classes in a system may impact Propagation Cost [29]. Therefore, lines of code (LOC) and the number of classes were gathered for each version as well.

The collected data was analyzed by comparing the number of detected architectural smells in each project to Propagation Cost, Decoupling Level, and code coverage, to identify any correlations. The architectural smell data for each project was analyzed using the Shapiro-Wilks normalcy test and found not to be normally distributed. Therefore, correlations were measured using Spearman's correlation coefficient ρ[8]. As per convention, the p-value must be less than .05 to be statistically significant [4], [9], [12], [28], and $-0.1 > \rho < 0.1$ is treated as being too weak to be significant. As potential confounding variables, correlations with LOC and the number of classes were 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 was also performed on the package level, by analyzing all unique packages in each project. In addition, a comparative analysis was conducted between different types of architectural smells. To assess potential differences, the correlations of architectural smells associated with impaired modularity or testability were compared to those of the non-impairing groups of smells.

To further analyze the findings, resulting empirically observable patterns were compared with this study's propositions, and the findings from each case were compared and contrasted in a cross-case synthesis.

D. Artifact

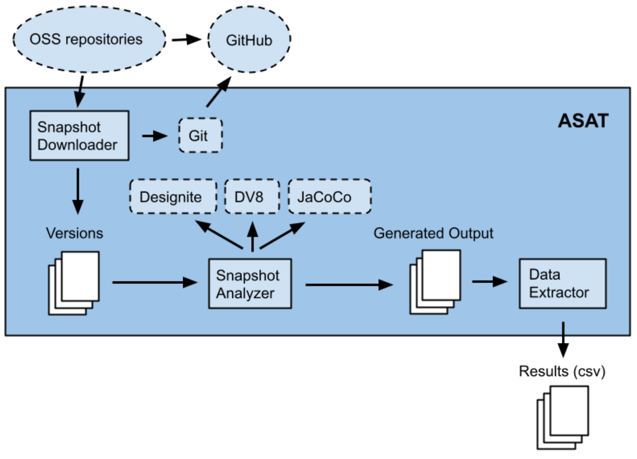


Fig. 4. Overview of the design of ASAT.

The principal artifact of this study, hereafter referred to as ASAT (Architectural Smell Analysis Tool), served as the primary means of data collection and analysis for this study. As shown in Figure 4, ASAT consists of three distinct modules:

1) **Snapshot Downloader:** This module is responsible for methodically retrieving versions at 4-week intervals from the inception to the most recent commit of the chosen GitHub repositories.
2) **Snapshot Analyzer:** This module serves to integrate three tools, namely Designite, JaCoCo, and DV8. Designite is utilized within ASAT to detect and measure different architectural smells within each version. JaCoCo is integrated within ASAT to measure testability in the form of code coverage. Finally, DV8 is used to measure the modularity of the versions by measuring their Decoupling Level and Propagation Cost. The generated output consists of files generated by each integrated tool, mostly in CSV format.
3) **Data Extractor:** The final module in ASAT extracts the relevant data from the output generated by the Snapshot Analyzer and compiles it into structured CSV files for subsequent inspection and analysis.

ASAT was instrumental in answering the research questions posed in this study, enabling an efficient and objective assessment of the relationships between architectural smells, modularity, and testability in software projects. The code for the Architectural Smell Analysis Tool can be accessed via GitHub[9]. All generated data is available in a reproduction package on GitHub as well[10].

VI. RESULTS

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

A. The Relationship between Architectural Smells and Project-level Modularity (RQ1a)

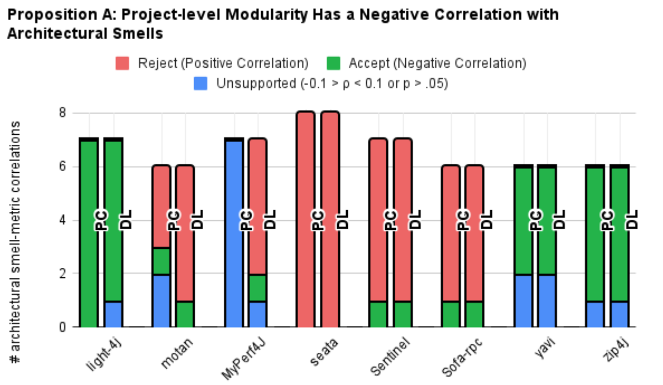


Fig. 5. Comparison of project-level correlations between smells and the metric Propagation Cost (PC) and Decoupling Level (DL). Shows negative correlations (Accept); positive correlations (Reject); and statistically insignificant correlations, where either $-0.1 > \rho < 0.1$ or $p > .05$ (Unsupported).

Figure 5 shows the results of analyzing the correlations between architectural smells and the modularity metrics Propagation Cost and Decoupling Level. For most cases, there was not a negative correlation between modularity and architectural smells—only three out of eight cases (light-4j, yavi, and zip4j)

[8]It is important to note that a negative 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 correlation" is used interchangeably with "negative relationship," indicating that the involved quality attribute decreases as the other variable (e.g. smell or size) increases. Since the negative relationship with Propagation Cost results in positive ρ values, the negative relationships with Decoupling Level and code coverage are also presented as positive ρ values to facilitate analysis.

[9]https://github.com/albe2003miun/ASAT

[10]https://github.com/albe2003miun/ASAT-reproduction-package

showed a mostly negative correlation. One exception was the Dense Structure smell which showed a negative correlation in all cases where it was detected except a single one: seata.

The majority of cases did not exhibit negative correlations between architectural smells and modularity at the project level. The Dense Structure smell was an exception, showing the opposite result.

As a rival explanation, proposition D proposes that the negative relationship between architectural smells and modularity can be explained by size instead, whereby the decrease in modularity and the increase in smells both more strongly correlate with 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 correlation 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 positive correlation between size and modularity is stronger. In the remaining results, only a single case (yavi) clearly accepts proposition D for modularity and size, while the remaining three cases (light-4j, motan, and zip4j) show mixed results.

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

The majority of cases did not have a stronger negative correlation between project-level modularity and size than architectural smells. In large part, this was because even if there was a positive correlation between architectural smells and modularity, the positive correlation between size and modularity was stronger. Also, architectural smells correlated more strongly with size than modularity, with the exception of the Dense Structure Smell.

B. The Relationship between Architectural Smells and Project-level Testability (RQ1b)

Analyzing the correlations between architectural smells and code coverage showed a negative relationship in most cases, as shown in Figure 6. Two cases, sofa-rpc and zip4j, showed a positive relationship instead. These two cases were the only ones that showed a trend of code coverage increasing as the projects expanded, instead of decreasing. Another outlier was the Dense Structure smell that only showed a negative correlation with testability in one case—light-4j. In all other cases, it showed a positive correlation or was not statistically significant ($p > .05$).

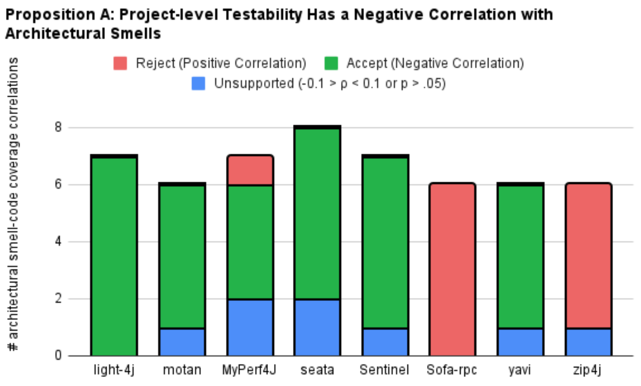


Fig. 6. Comparison of project-level correlations between smells and the code coverage. Shows negative correlations (Accept); positive correlations (Reject); and statistically insignificant correlations, where either $-0.1 > \rho < 0.1$ or $p > .05$ (Unsupported).

At the project level, six out of eight cases had stronger negative correlations between testability and architectural smells than size. However, for the Dense Structure smell the same result was only seen in a single case.

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

Comparing the strength of negative correlations between architectural smells and testability and positive correlations between architectural smells and size, all cases showed a stronger correlation to size than to testability. The only smell that showed a stronger correlation with testability was Dense Structure in a single case: MyPerf4J.

Overall, testability had a stronger negative correlation with size than with architectural smells at the project level. However, the opposite was seen in cases that had a positive correlation between testability and architectural smells. For architectural smells, all cases unanimously showed a stronger correlation between architectural smells and size than architectural smells and testability.

C. The Relationship between Architectural Smells and Testability in Packages (RQ1c)

As shown in Figure 7, most detected architectural smells in packages did not show a correlation with the code coverage in those packages[11]. Out of the eight cases, six predominately did not establish correlations for the majority of smells. Out of the remaining two cases, one case—motan—showed 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, showed a tie between smells with mostly established and mostly unsupported correlations, with two out of three mostly established correlations (Feature Concentration and Scattered Functionality) showing mainly negative correlations.

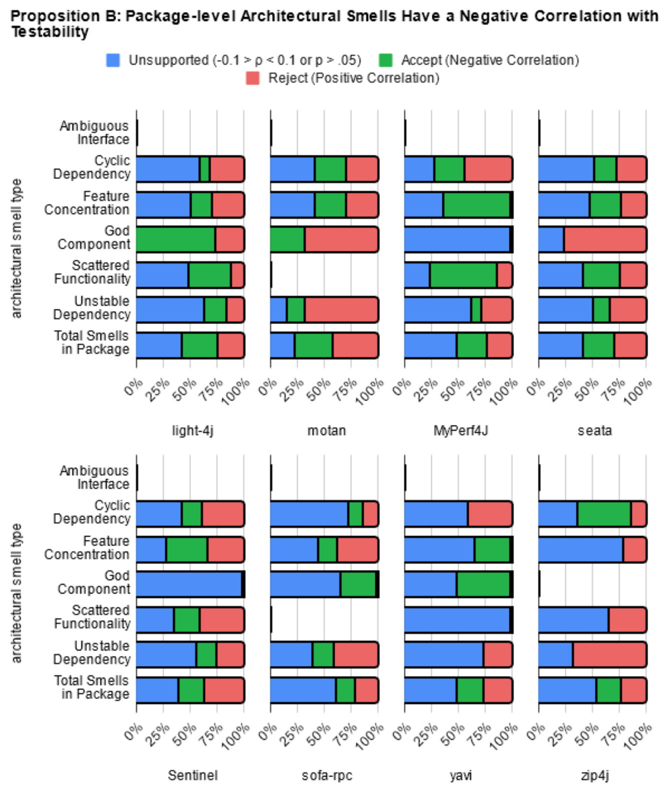


Fig. 7. Comparison of package-level correlations between smells and the code coverage. Shows % of packages with negative correlations (Accept); positive correlations (Reject); and statistically insignificant correlations, where either $-0.1 > \rho < 0.1$ or $p > .05$ (Unsupported).

Putting unsupported correlations aside, the two cases where code coverage increased with size (sofa-rpc and zip4j) showed a majority of positive correlations. However, unlike at the project-level, the opposite result was not observed in the remaining cases. In total, five of the eight cases had mostly positive correlations. In contrast, only two cases showed a majority of negative correlations (light-4j and MyPerf4J), and yavi was a tie. It is worth noting, however, that the cases zip4j and yavi presented 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 found in Table II.

In most cases, the majority of correlations between architectural smells and testability in packages were statistically insignificant.

As a rival explanation, the negative correlations between testability and architectural smells in packages were compared to the negative correlations between testability and size, and the positive correlations between architectural smells and size. The results show that the negative correlations were stronger with testability and size than with testability and smells in all cases. The same was true for all types of smells, but the relationship between Cyclic Dependency and testability compared most favorably to size, beating it in three cases and tieing in one.

In addition, the positive relationship between architectural smells and size was clearly stronger than the negative relationship between architectural smells and testability in all cases. There were only two outliers among the detected smells, Feature Concentration in yavi and Cyclic Dependency in zip4j.

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

D. Comparing Relationships between Different Types of Architectural Smells and Project-level Modularity (RQ2a)

The purpose of this RQ was to test whether architectural smells that are said to specifically impair modularity exhibit a stronger negative correlation with it than other smells (proposition C). The results are shown in Table III, where a higher ρ signifies a stronger negative correlation 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, two of the smells in the impairing group (God Component and Scattered Functionality) exhibit a lower correlation with modularity than the non-impairing smells in most cases. 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[12]. Cyclic Dependency showed a stronger correlation in all cases except two—seata and sofa-rpc—where it was slightly lower. Ambiguous Interface was only present in seata, but showed a 10% and 23% percent higher correlation

[11]No correlation means that either $p > .05$ or $-0.1 > \rho < 0.1$. If a detected smell is a constant, i.e. its value does not change across all versions, it is not possible to calculate a ρ value, so it is not included.

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

TABLE III
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

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. PC: Propagation Cost, DL: Decoupling Level.

TABLE IV
PROJECT-LEVEL COMPARISON OF THE ρ DIFFERENCE OF ARCHITECTURAL SMELLS THAT IMPAIR TESTABILITY AND THEIR COUNTERPARTS

Smell	light-4j	motan	MyPerf4J	seata	Sentinel	sofa-rpc	yavi	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

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.

TABLE V
PACKAGE-LEVEL COMPARISON OF THE ρ DIFFERENCE OF ARCHITECTURAL SMELLS THAT IMPAIR TESTABILITY AND THEIR COUNTERPARTS

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

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.

with Propagation Cost and Decoupling Level respectively, relative to the group of non-impairing smells.

At the project level, the modularity-impairing architectural smells Ambiguous Interface, Cyclic Dependency, and Dense Structure all showed stronger negative correlations than their non-modularity-impairing counterparts. In particular, Dense Structure showed an especially strong negative correlation with modularity. The two remaining modularity-impairing smells, God Component and Scattered Functionality showed weaker negative correlations than the non-impairing group of smells in most cases.

E. Comparing Relationships between Different Types of Architectural Smells and Project-level Testability (RQ2b)

As shown in Table IV, the group mean of architectural smells that are said to impair testability (Dense Structure, God Component, and Scattered Functionality) showed a stronger negative correlation than non-impairing smells (Ambiguous Interface, Cyclic Dependency, Feature Concentration, and Unstable Dependency) in five out of eight cases. Two of the three remaining cases—motan and light-4j—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 was 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 the majority of cases.

At the project level, two of the three architectural smells that are said to impair testability—God Component and Scattered Functionality—showed stronger negative correlations than their non-impairing counterparts. The third smell—Dense Structure—showed the opposite result in most cases.

F. Comparing Relationships between Different Types of Architectural Smells and Testability in Packages (RQ2c)

The group mean of testability-impairing smells showed a stronger negative correlation with testability in five out of eight cases, thus accepting proposition C. The remaining cases that rejected proposition C were the two largest projects containing the most packages (seata and Sentinel), and the smallest project with the least packages (zip4j). It is also noteworthy that there was no overlap between which cases rejected proposition C at the project-level (RQ2b) and at the package-level (RQ2c). The results are displayed in Table V.

As for individual smells[13], God Component had a higher correlation with testability than non-impairing smells in four out of five cases where it was present. Similarly, Scattered Functionality showed a higher correlation in three out of five cases where it was present. The finding was in line with the architectural smells on the project-level, but there was no overlap between which cases did not show a higher correlation for these smells. One of the non-testability-impairing smells—Feature Concentration—tied in the number of stronger and weaker negative correlations when compared to the group of testability-impairing smells. Notably, considering all results for both modularity and testability at both the project and package level, it was the only non-impairing smell that did not exhibit a distinctly weaker negative correlation.

Within packages, both God Component and Scattered Functionality, which are said to impair testability, showed stronger negative correlations than their non-impairing counterparts in the majority of cases.

VII. Discussion

Contrary to expectations, architectural smells did not show a negative correlation 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, Decoupling Level should not be sensitive to size in this way, yet it also showed a positive relationship with size in a majority of cases. Indeed, the positive correlations between modularity and size were found to be stronger than modularity and architectural smells in general. In addition, architectural smells were found to be more strongly correlated to size than to modularity. In other words, while both architectural smells and modularity tend to increase as a project grows in size, that positive correlation is likely a spurious relationship that can be explained by the increasing size.

It is also important to note that there was a single smell—Dense Structure—that stood out from the rest in RQ1a. It showed a negative correlation with modularity in six out of seven cases where it was detected and a stronger correlation with modularity than size in four out of seven cases. This is no surprise, given how Dense Structure is measured. Unlike the other detected architectural smells, Dense Structure can only be detected at the project-level for all packages and is binary in that it is either detected or it isn't. As such, in order to be able to meaningfully measure correlations with Dense Structure, the strength of the smell—measured in average degree—was used instead of the number of detected instances. The average degree is a measure of the average amount of dependencies between packages in a project. This makes it similar to Propagation Cost, which is calculated from dependencies (including transitive dependencies) between classes. Therefore, both measure coupling, albeit in different ways and at levels of granularity.

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

On the other hand, the results of RQ1b showed that there is a negative correlation between architectural smells and code coverage at the project-level. There were two cases that showed the opposite results. In both of these cases code coverage increased as the size of the project increased. This resulted in positive correlations between smells and code coverage, but as with modularity in RQ1a, these correlations were weaker than the positive correlations between code coverage and size. However, the same was true for the negative correlations observed in other cases as well—they were weaker than the negative correlations with size. Furthermore, architectural smells also showed a weaker 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 be contributing to this pattern. This assumption gains credibility considering the inconsistent positive or negative correlations 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 those versions—making it difficult to establish a correlation. A similar pattern can be observed in smaller packages as well, 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 correlation with code coverage was found to be stronger with size than with smells. In other words, there does not seem to be a negative correlation 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 correlations 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, it does not preclude a common confounding factor. We have shown that size was a stronger factor than smells in both the negative correlation with testability and the positive correlation with modularity. But it is possible that there is 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, quantifying technical debt is difficult, given its definition as the "invisible result of past decisions" [24]. 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 [20]. While this seems to be in line 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.

The purpose of RQ2 was to test whether architectural smells that are said to impair specific sub-characteristics of maintainability actually have a stronger negative relationship with said characteristics than their counterparts. 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 [1]. As shown in the results of RQ2, this turned out to be mostly the case. In total, the smells said to impair modularity or testability displayed a higher negative correlation for seven out of the ten smells investigated at both package and project levels. Dense Structure had an especially high affinity with modularity, the possible reason for which was 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 relationship with modularity but did do so 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 relationship with testability in packages when compared to the group of impairing smells. But overall, there were no non-impairing smells that showed a pattern of stronger negative correlations than the group of affecting smells.

As far as we are aware, this is the first empirical result that shows 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 correlations (if discounting statistically insignificant results). This result is a large departure from previous research, which did not make any distinction between the type of smell [9], or did not see any significant difference in impact on the number of source code changes based on the type of smell [4]. The results lend credence to the notion that different smells can be used as indicators of potential quality issues. For example, the smells God Component and Scattered Functionality act as stronger indicators of reduced testability than their counterparts. This knowledge may help give clues to developers about which parts of their systems have specific issues, or where to prioritize refactorings.

Still, it is important to acknowledge the notion that not every instance of an architectural smell is necessarily harmful, but rather requires individual analysis. Alternatively, some smells might be implemented on purpose as part of a solution, such as in certain design patterns. In this study, this has not been taken into consideration, since it would require considerable effort. Rather, we have looked for trends by analyzing types of architectural smells as a group, as opposed to analyzing each individual instance of a smell.

A. *Threats to Validity*

This study does not try to establish any causal relationships, so threats to internal validity are not applicable.

Construct validity

Construct validity is concerned with the suitability and accuracy of measurements used in the study. One threat is the interval between versions. We chose four weeks since it had been used in previous studies with good results, and because employing a lower interval would have made it computationally infeasible to analyze the versions without reducing the number of projects. However, in projects with a lot of activity, it is possible that using a lower interval would result in more detailed results. Conversely, two of the included projects showed lower activity than desired, having more than four weeks between some commits. This was an unfortunate side-effect of having to prioritize projects where it was possible to get uniform test coverage data. In addition, there were a few versions that had to be discarded due to build issues which made 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 Figure 3, the largest variance is in light-4j by 8.56%. However, this was a very gradual decrease over six and a half years, where the number of tests go from less than 100 to over 800. Therefore, there was not a large fluctuation between versions, but as there were failed tests the code coverage data is not perfect. The same can be said for all included projects except yavi, which had a 100% test success rate across all versions.

The choice of Propagation Cost and Decoupling Level to measure modularity could also pose a threat. Both Propagation Cost and Decoupling Level tended to increase with project size. As discussed earlier, we theorize that this is simply a characteristic of large projects, where the number of classes increases more in proportion than the dependencies among them, resulting in an overall lower modularity. However, while these metrics give an estimate of the modularity of a project as a whole, it does not say anything about the modularity in smaller components such as modules or packages. Unfortunately, we were unable to find a means of quantifying modularity at these levels, which might otherwise have provided a more detailed analysis.

We believe that using code coverage to measure testability was apt since it is the practical application of the concept. Of course, code coverage can still 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 amount of developer effort that was needed to implement the tests. Without input from the developers themselves, it is impossible to conclude definitively whether architectural smells have an effect on the efficiency of writing tests.

External validity

External validity is concerned with the generalizability of the findings in the study. There are two threats concerning the cases that were included in this study. The first one is about the type of projects that were selected. While all projects are open-source and implemented in Java, we strove to select projects 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. The second threat concerns the number of cases. While eight cases are not low compared to many other recent studies about architectural smells[14], including more cases would certainly increase the generalizability of the findings. Both of these factors were largely influenced by the effort required to perform the dynamic code coverage analysis of each included project.

Another threat concerns the applicability of the presented findings on all architectural smells. The results in this study only apply to the architectural smells that were measured and do not make any conclusions about other smells that were not included or all architectural smells as a whole. In addition, the Ambiguous Interface smell was only present in a single case, so results concerning this smell should be treated with caution.

Reliability

Reliability is concerned with the replicability of the study. This study follows a well-established case study design framework by Yin [28] and is transparent about the methodology used to produce its results. In addition, the self-developed tool (ASAT) used in this study, as well as all collected data, is freely available online for study or replication.[15]. However, 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.

VIII. Conclusions

The main findings of this study show that: Firstly, there was not a negative relationship between project-level modularity and architectural smells as a whole in the majority of cases. However, the Dense Structure smell stood out as an exception, showing a strong negative correlation with modularity—even stronger than its correlation with size. Secondly, a negative correlation was observed between architectural smells and project-level testability. However, project size shows a stronger relationship with both, suggesting it may be a spurious association. Thirdly, the analysis of package-level smells revealed that they generally did not exhibit significant correlations with testability, and when correlations were present, they were not predominantly negative. Fourthly, seven out of the ten examined architectural smells demonstrated a stronger negative correlation with the specific quality attributes they were claimed to impair when compared to their counterparts. Three out of five smells associated with modularity impairment—Ambiguous Interface, Cyclic Dependency, and Dense Structure—exhibited higher negative correlations at the project-level. For testability, the associated smells God Component and Scattered Functionality displayed higher negative correlations at both the project and package levels. The implications of this are twofold. Firstly, for software developers, it can serve as a guide to better understand the significance of specific architectural smells in their own projects. Secondly, for researchers, it provides empirical evidence that substantiates parts of the existing knowledge base about the effects of specific architectural smells, which has to date predominantly relied on intuition and individual experience.

A future research recommendation is to explore ways to quantify not only package-level modularity but also additional sub-characteristics of maintainability, in order to examine their relationships with different architectural smells. In particular, to investigate whether these quality attributes also have a stronger negative relationship with the smells that are said to impair them. Moreover, we have theorized that the observed increase in architectural smells and modularity might be a natural consequence of projects growing in size. But it is possible that there is another lurking factor that is responsible for the observed correlations between architectural smells and modularity/testability, such as technical debt or even age. Investigating whether there are alternative factors that can better explain the correlations between architectural smells and quality attributes would certainly be an interesting subject for research.

[14]For example, Le et al. [9] also used eight cases in their study.

[15]The tool is avaiable at: https://github.com/albe2003miun/dt133g_project. The reproduction package is available at: https://github.com/albe2003miun/dt133g-project-reproduction-package.

References

[1] P. Rachow and M. Riebisch, "An architecture smell knowledge base for managing architecture technical debt," in *Proceedings of the International Conference on Technical Debt*, 2022, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3524843.3528092

[2] F. Tian, P. Liang, and M. A. Babar, "How developers discuss architecture smells? an exploratory study on stack overflow," in *2019 IEEE international conference on software architecture (ICSA)*. IEEE, 2019, pp. 91–100. [Online]. Available: https://doi.org/10.1109/ICSA.2019.00018

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

[4] D. Sas, P. Avgeriou, I. Pigazzini, and F. Arcelli Fontana, “On the relation between architectural smells and source code changes,” *Journal of Software: Evolution and Process*, vol. 34, no. 1, p. e2398, 2022. [Online]. Available: https://doi.org/10.1002/smr.2398

[5] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Identifying architectural bad smells,” in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258. [Online]. Available: https://doi.org/10.1109/CSMR.2009.59

[6] F. Arcelli Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi, “Are architectural smells independent from code smells? an empirical study,” *Journal of Systems and Software*, vol. 154, pp. 139–156, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.04.066

[7] T. Sharma and M. Kessentini, “Qscored: A large dataset of code smells and quality metrics,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 590–594. [Online]. Available: https://www.doi.org/10.1109/MSR52588.2021.00080

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

[9] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, “An empirical study of architectural decay in open-source software,” in *2018 IEEE International conference on software architecture (ICSA)*. IEEE, 2018, pp. 176–17 609. [Online]. Available: https://doi.org/10.1109/ICSA.2018.00027

[10] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, “Software architecture degradation in open source software: A systematic literature review,” *IEEE Access*, vol. 8, pp. 173 681–173 709, 2020.

[11] D. Sas, P. Avgeriou, R. Kruizinga, and R. Scheedler, “Exploring the relation between co-changes and architectural smells,” *SN Computer Science*, vol. 2, pp. 1–15, 2021. [Online]. Available: https://doi.org/10.1007/s42979-020-00407-5

[12] I. Pigazzini, F. A. Fontana, and B. Walter, “A study on correlations between architectural smells and design patterns,” *Journal of Systems and Software*, vol. 178, p. 110984, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2021.110984

[13] J. A. Diaz-Pace, A. Tommasel, I. Pigazzini, and F. A. Fontana, “Sen4smells: A tool for ranking sensitive smells for an architecture debt index,” in *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/ARGENCON49523.2020.9505535

[14] “Understand, refactor, and control software architecture,” accessed Nov. 11, 2022. [Online]. Available: https://www.lattix.com/

[15] “Software intelligence automated: Cast,” accessed Nov. 11, 2022. [Online]. Available: https://www.castsoftware.com/

[16] B. Chantian and P. Muenchaisri, “A refactoring approach for too large packages using community detection and dependency-based impacts,” in *Proceedings of the 1st World Symposium on Software Engineering*, 2019, pp. 27–31. [Online]. Available: https://doi.org/10.1145/3362125.3362132

[17] “Iso/iec 25010:2011,” accessed Jan. 06, 2023. [Online]. Available: https://www.iso.org/standard/35733.html

[18] H. Mumtaz, P. Singh, and K. Blincoe, “A systematic mapping study on architectural smells detection,” *Journal of Systems and Software*, vol. 173, p. 110885, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110885

[19] U. Azadi, F. Arcelli Fontana, and D. Taibi, “Architectural smells detected by tools: a catalogue proposal,” in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 88–97. [Online]. Available: https://www.doi.org/10.1109/TechDebt.2019.00027

[20] P. Gnoyke, S. Schulze, and J. Krüger, “An evolutionary analysis of software-architecture smells,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 413–424. [Online]. Available: https://doi.org/10.1109/ICSME52107.2021.00043

[21] B. Mehboob, C. Y. Chong, S. P. Lee, and J. M. Y. Lim, “Reusability affecting factors and software metrics for reusability: A systematic literature review,” *Software: Practice and Experience*, vol. 51, no. 6, pp. 1416–1458, 2021. [Online]. Available: https://doi.org/10.1002/spe.2961

[22] R. C. Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*, 1st ed. Prentice Hall Press, 2017.

[23] W. Cunningham, “The wycash portfolio management system,” *SIGPLAN OOPS Mess.*, vol. 4, no. 2, p. 29–30, dec 1992. [Online]. Available: https://doi.org/10.1145/157710.157715

[24] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012. [Online]. Available: http://doi.org/10.1109/MS.2012.167

[25] A. Martini, J. Bosch, and M. Chaudron, “Architecture technical debt: Understanding causes and a qualitative model,” in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014, pp. 85–92. [Online]. Available: https://doi.org/10.1109/SEAA.2014.65

[26] M. G. Stochel, P. Chołda, and M. R. Wawrowski, “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)*, 2020, pp. 367–375. [Online]. Available: https://doi.org/10.1109/SEAA51224.2020.00067

[27] I. Pigazzini, F. A. Fontana, and B. Walter, “A study on correlations between architectural smells and design patterns,” *Journal of Systems and Software*, vol. 178, p. 110984, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2021.110984

[28] R. K. Yin, *Case study research and applications*, 6th ed. Thousand Oaks, CA: SAGE Publications, Jan. 2018.

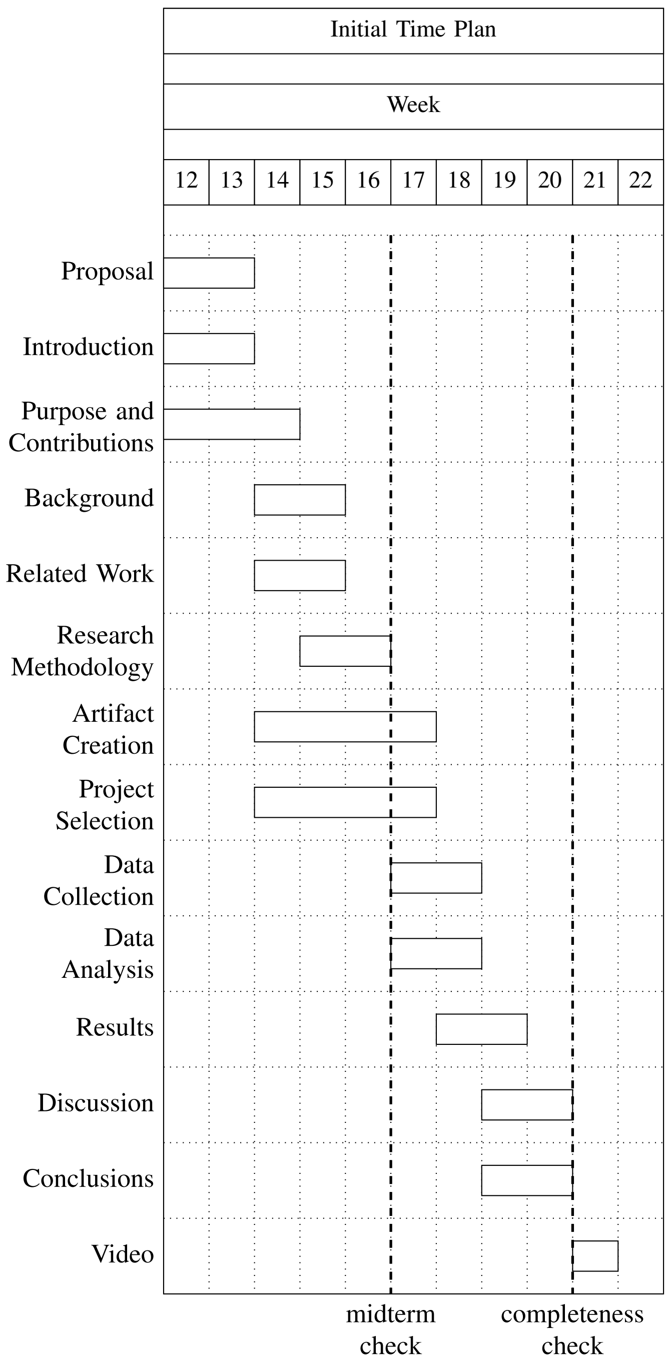
[29] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, “Decoupling level: A new metric for architectural maintenance complexity,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 499–510. [Online]. Available: https://doi.org/10.1145/2884781.2884825

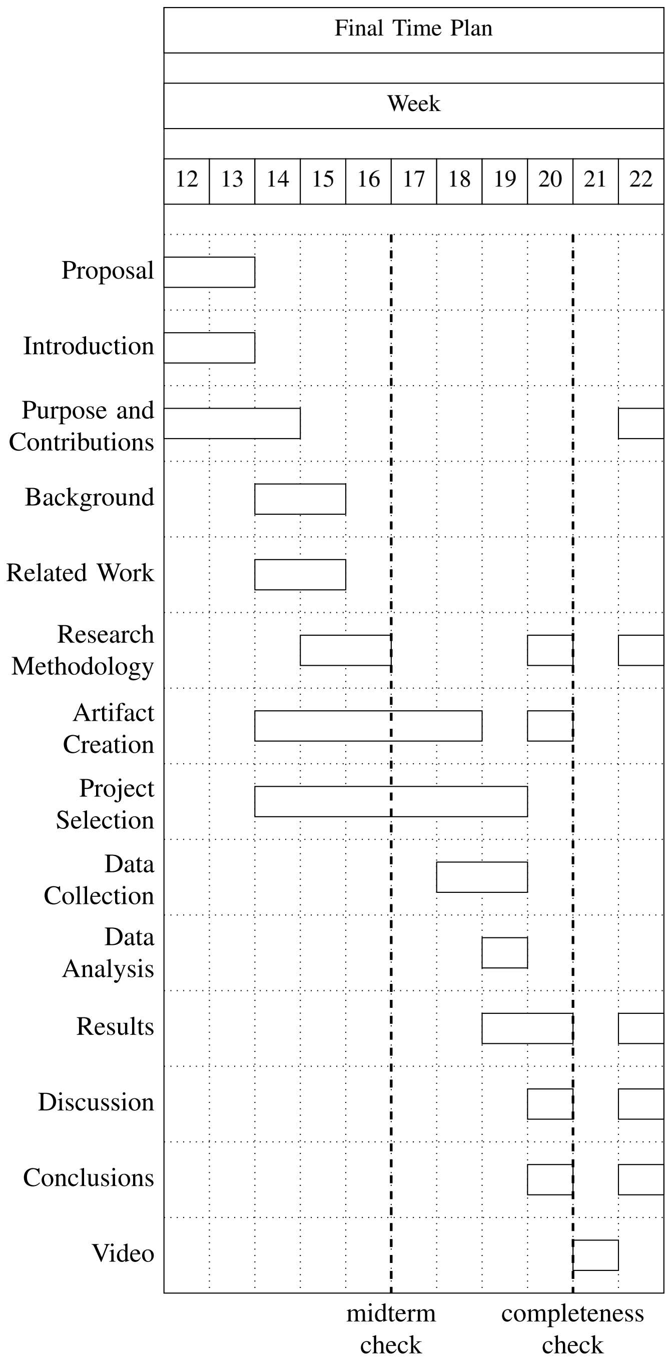
[30] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: An empirical study of open source and proprietary code,” *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006. [Online]. Available: https://doi.org/10.1287/mnsc.1060.0552

[31] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele, “Experiences applying automated architecture analysis tool suites,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 779–789. [Online]. Available: https://doi.org/10.1145/3238147.3240467

[32] T. Yasi and J. Qin, “Automatic building of java projects on github: A study on reproducibility,” 2022.

Appendix 1: Time Plan





Appendix 2: Contributions

The contributions can be summarized as follows:

- Alexander Berglund was the primary contributor to the artifact and the Artifact section in the Research Methodology.
- Simon Karlsson was the primary contributor to the Introduction, Purpose and Contributions, Background, Related Work, and Research Methodology sections.
- Both researchers made contributions to the remaining areas, including the proposal, project selection, data collection, and data analysis, as well as the Results, Discussion, and Conclusions sections.

It should be noted that, naturally, there was collaboration and contributions both within and beyond these specific areas.