On the correlation between Architectural Smells and Static Analysis Warnings

Matteo Esposito[a], Mikel Robredo[a], Francesca Arcelli Fontana[b], Valentina Lenarduzzi[a]

[a] *University of Oulu, Finland*
[b] *University of Milano-Bicocca, Italy*

Abstract

Background. Software quality assurance is essential during software development and maintenance. Static Analysis Tools (SAT) are widely used for assessing code quality. Architectural smells are becoming more daunting to address and evaluate among quality issues.
Objective. We aim to understand the relationships between Static Analysis Warnings (SAW) and Architectural Smells (AS) to guide developers/maintainers in focusing their effort on SAWs more prone to co-occurring with AS.
Method. We performed an empirical study on 103 Java projects totaling 72 million LOC belonging to projects from a vast set of domains, and 785 SAW detected by four SAT, Checkstyle, Findbugs, PMD, SonarQube, and 4 architectural smells detected by ARCAN tool. We analyzed how SAWs influence AS presence. Finally, we proposed an AS remediation effort prioritization based on SAW severity and SAW proneness to specific ASs.
Results. Our study reveals a moderate correlation between SAWs and ASs. Different combinations of SATs and SAWs significantly affect AS occurrence, with certain SAWs more likely to co-occur with specific ASs. Conversely, 33.79% of SAWs act as "healthy carriers," not associated with any ASs.
Conclusion. Practitioners can ignore about a third of SAWs and focus on those most likely to be associated with ASs. Prioritizing AS remediation based on SAW severity or SAW proneness to specific ASs results in effective rankings like those based on AS severity.

Keywords: Static Analysis Tool, Software Quality Warnings, Architectural Smells, Technical Debt

1. Introduction

The risk of service degradation and disruption for critical services is exponentially increasing due to the pervasive nature of computing and the widespread adoption of digital services in all areas of daily life [1]. To create a common definition for quality issues, the literature provides the concept of *smells* [2], referring to quality issues in code [3]. These smells are usually categorized as code smells, low level, coding issues [4, 2], and, recently, in architectural smells, higher abstract issues that concern the software architecture and the software components [5]. More specifically, architectural smells (AS) are "architectural decisions that negatively impact system internal quality" [6], which can occur simply by "applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity" [6]. High-level quality issues are more difficult to spot given the non-direct, and evident connection, to low-level coding issues [7]. Therefore, recent studies stressed the role of AS in architectural degradation, thus hastening architectural technical debt accumulation [8, 9, 10].

A popular technique to detect quality issues is to leverage Static Analysis Tools (SAT) [11, 12] which analyze the source code without executing it (thus static analyzers), and produce a report containing all the violations, i.e., Static Analysis Warnings (SAW) that the code present. More specifically, SAWs, refer to code violations with a lower-level granularity, compared to AS. Therefore, SAWs do not consider the software architecture of the system under development, commonly focusing on class or method levels [13]. While multiple studies successfully exploited SATs to remediate common quality issues [13, 14, 3, 15, 16], only a few studies focused on developing SATs to detect ASs [5]. Moreover, some studies have investigated the interrelations between code anomalies [17, 18] or code smells [19] and AS [17, 18]. However, to our knowledge, no previous study has analyzed the correlations between SAW and AS. Furthermore, to ensure the safety and stability of services, during development and maintenance, AS remediation should be prioritized [9, 10]. Hence, the portion of code affected by AS is the primary remediation target to prevent critical quality issues.

Therefore, the need to investigate how practitioners and researchers can exploit SAWs to detect and remediate AS is evident. Hence we designed and performed a large

Email addresses: matteo.esposito@oulu.fi (Matteo Esposito), mikel.robredomanero@oulu.fi (Mikel Robredo), francesca.arcelli@unimib.it (Francesca Arcelli Fontana), valentina.lenarduzzi@oulu.fi (Valentina Lenarduzzi)

empirical study on the possible correlations between four AS and SAW detected by four tools largely adopted by developers [13].

Hence, this study assesses the correlation between SAW and higher-level AS. We did not consider correlations between defects and SAW, as these have been extensively studied in the literature. Additionally, we investigate the correlation of SAW and AS, and how SAW can be utilized to prioritize AS. Discovering a correlation between AS and SAW would enable the use of existing SAT as an effective tool for detecting issues in software architecture. For practitioners, linking SAW to architectural smells can reduce the need for refactoring, thereby saving both cost and effort [20, 21]. Therefore, we can summarize our main contributions as follows:

- We performed the first, according to our knowledge, large-scale study, i.e., 112 projects, on the correlation of ASs and SAWs. Other studies have considered correlations with more general architectural problems [18, 17] or the correlation between AS and code smells [19].
- We investigated the contribution of the most correlated SAWs with ASs.
- We proposed and evaluated AS remediation effort prioritization via SAWs severity.

Our study focused on 112 Java projects from the Qualitas Corpus dataset (QCD) [22], totaling 72 million lines of code analysis and 66,036 packages, 808,208 classes, and 88,460 interfaces examined across various contexts, including development platforms, multimedia, gaming, middleware, compilers, and testing frameworks. We considered a large set of warnings (1142) and four AS based on dependency issues that can critically influence the quality of a software project and its progressive architecture degradation [23].

Our findings reveal a moderate correlation between SAWs and ASs. Moreover, different combinations of SATs and SAWs significantly influence the occurrence of ASs. Therefore specific SAWs are more prone to co-occur with specific ASs. Conversely, we also identified SAWs acting as "healthy carriers," i.e., not co-occurring with any ASs. Healthy carriers represent 33.79% of our dataset. Our study shows that practitioners can safely drop a third of the SAW in our context and focus on the most AS-prone to address AS without prior AS severity or discovery knowledge effectively. Our study highlighted that prioritizing AS remediation efforts based on SAW severity, or SAW proneness to specific AS with a limited inspection window, results in rankings similar to the optimal one, i.e., ordered by AS severity.

Therefore, we suggest that SAWs can influence AS's presence. Prioritizing SAWs based on occurrence probability effectively guides remediation. Future research should improve the detection of AS-prone SAWs and explore deeper connections between SAWs and AS for better understanding and prioritization of remediation efforts.

Paper Structure. Section 2 describes the background to which our paper is based. In Section 3, we present the study design. In Section 4, we show the results obtained and discuss them in Section 5. Section 6 focuses on threats to the validity of our study. Section 7 discusses related work and in Section 8, we draw conclusions and outline future research directions.

2. Background

2.1. Static Analysis Tool

This section introduces the tools we employed in our study and the motivation for their selection. SAT analyzes software without running it by inspecting the source code, thus discovering potential quality issues in the code base [8]. Recently, the ease-to-useness of SATs and the subsequent inclusion in CI practices [24] are boosting their popularity. Among the SAT available, we selected the most widely used according to state of the art in detecting SAW [11, 13] such as Sonarqube, Checkstyle, Findbugs, and PMD. While according to architectural issues, our study focuses on architectural smells (AS). Therefore, we also included ARCAN based on our previous experience [5]. In Table 1, we summarized the characteristics of each of the following SATs.

More specifically, regarding SATs, **Checkstyle**[1] is designed to asses Java code quality. The tool uses "rules" according to a set of *checks* for analyzing the code base. Checkstyle has two predefined rule configurations: Google Java Style and Sun Java Style. Static violations of the checks are grouped under two severity levels: error and warning.

FindBugs[2] examine Java bytecode. The tool detects *bug patterns* which are caused by, but not limited to, difficult language features and misunderstood API features. Bug patterns are ranked on an ordinal scale from one to 20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group.

PMD[3] can inspect eight programming languages, including Java. PMD uses a set of *rules* to assess code quality. There are a total of 33 different rule-set configurations for Java projects. Rules are classified into 8 categories. The tool measures rule violations on an ordinal scale of one to five. One is associated with the most severe violations, and 5 is the least severe.

SonarQube[4] is one of the most common SATs for detecting code quality issues. SonarQube measures several aspects of the source code, such as several lines of code and

[1] `https://checkstyle.org`
[2] `http://findbugs.sourceforge.net`
[3] `https://pmd.github.io/`
[4] `http://www.sonarsource.org/`

SAW	Checkstyle	Findbugs	PMD	SonarQube
Warnings Types	14 types	9 types	8 types	3 types
Total Warnings	173	424	305	413
Types	Annotations, Block Checks, Class Design, Coding, Headers, Imports, Javac Comments, Metrics, Miscellaneous, Modifiers, Naming Conventions, Regexp, Size Violations, Whitespace	Bad practice, Correctness, Experimental, Internationalization, Malicious code vulnerability, Multithreaded correctness, Performance, Security, Dodgy code	Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance, Security	Bugs, Code Smells, Vulnerabilities
Severity Levels	Error, Ignore, Info, Warning	1-20 ranking: Scariest, Scary, Troubling, Concern	Change required, Change suggested, Change optional, Change highly recommended, Change highly optional	Blocker, Critical, Major, Minor, Info

Table 1: Summary of Static Analysis Tools Warning (SAW)

code complexity, and verifies code compliance with a specific set of *coding rules* defined for most common programming languages. Violations of coding rules are reported as *issues* in SonarQube. The estimated time to remove these issues, i.e., remediation effort, is used to compute the remediation cost. SonarQube analysis encompasses quality aspects such as reliability, maintainability, and security. Reliability rules, also named *Bugs*, create quality issues that "represent something wrong in the code" and that will soon be reflected in a bug. *Code smells* are considered "maintainability related issues" in the code that decrease code readability and modifiability. It is important to note that the term "code smells" adopted in SonarQube does not refer to the commonly known code smells term defined by Fowler et al. [4], but to a different set of rules.

On the other hand, we employed **ARCAN** to detect different AS based on dependency issues by computing different metrics proposed by Martin [25], such as those related to instability issues. We refer to software instability as the inability to make changes without impacting the entire system or a large part.

The tool relies on graph database technology to perform graph queries, which lets on higher scalability during the detection process and management of many different kinds of dependencies. The detailed detection techniques for AS have been described in previous works [5]. Moreover, the tool allows the computation of an architectural debt [26] index based on the number of AS detected in a project and the criticality of the AS, evaluated according to each type of AS. The tool's results were validated on ten open source systems and two industry projects with a high precision value of 100% in the results and 63% recall [5]. Moreover, the results of ARCAN were validated using practitioners' feedback on four industry projects [27]. In our study, we have considered the following AS detected by the tool:

1. *Unstable Dependency (UD)*: describes a subsystem (component) that depends on other less stable subsystems than itself [28]. This may cause a ripple effect of changes in the system [5]. Detected in packages.
2. *Hub-Like Dependency (HD)*: This smell arises when an abstraction has (outgoing and ingoing) dependencies on a large number of other abstractions [29]. Detected in classes and packages.
3. *Cyclic Dependency (CD)*: refers to a subsystem (component) that is involved in a chain of relations that break the desirable cyclic nature of a subsystem's dependency structure. The subsystems involved in a dependency cycle can hardly be released, maintained, or reused in isolation. Detected in classes and packages. The *Cyclic Dependency* AS is detected according to different shapes [30] as described in [5].

We focused on the above AS based on dependency issues since dependencies are very important in software architecture. Components with many dependencies can be considered more critical and expensive to maintain. Moreover, we defined as "healthy" the instances without AS, i.e., a software entity affected by one or more SAWs but without any of the ASs considered in our study.

2.2. Effort Aware Metrics

Çarka et al. [31] investigated effort-aware metrics (EAM) to improve defect prioritization. In our study, we take inspiration from the definition of PofB and PopT to create three different prioritization rankers for AS. We then evaluate our ranker in a way that is similar to the interpretation of the PofB results. Therefore, we briefly discuss EAMs; according to Çarka et al. [31], two different EAMs exist: normalized by size or not by size. The most known not-normalized EAM is called *PofB* [32, 33, 34, 35] PofB, or Proportion of Defects Identified by the top x% of the codebase, is a measure derived from a prediction model's rankings. It signifies the percentage of defective entities discovered when analyzing the initial x% of the code. Higher PofB indicates more effective rankings, thereby enhancing testing support. For instance, a PofB10 of 30% implies that 30% of defective entities were detected by analyzing 10% of the codebase according to the method's ranking.

Comparing a prediction model's ranking with a perfect one sheds light on its performance. Mende et al. [36], drawing inspiration from Arisholm et al. [37], introduced "Popt" to gauge this accuracy. It assesses how the prediction model deviates from perfection while outperforming random rankings. Popt quantifies the area Δopt between the ideal and predicted models. In the optimal scenario, instances are ordered by decreasing fault density, while in the predicted model, they're ordered by decreasing predicted defectiveness. The equation for Popt, $1 - \Delta opt$, indicates a larger value denotes a smaller gap between the ideal and predicted models [38].

Popt and PofB are distinct metrics, each capturing different facets of model accuracy. They employ different ranking methods: Popt ranks entities based on bug density (bug probability divided by entity size), while PofB ranks them according to bug probability. Consequently, classifiers ranked by Popt and PofB may differ. Popt, grounded in bug density, offers a more realistic perspective than PofB, which relies solely on probability. However, interpreting Popt is more challenging; a classifier with double the Popt value doesn't necessarily yield double the benefits to the user.

3. Empirical Study Design

This section presents the empirical study's goal, research questions, metrics, and hypotheses. Moreover, we describe the study context, the data collection, and the data analysis procedures. We designed and conducted our empirical study according to the guidelines defined by Wohlin et al. [39].

3.1. Goal, Research Questions, Metrics, and Hypothesis

We formalized the **goal** of this study according to the Goal Question Metric (GQM) approach [40] as follows:

The *goal* of our empirical study is to investigate the relationship between architectural smells and static analysis warnings in the context of software quality assurance. Our *perspective* is of practitioners and researchers seeking to assess the relationship between high-level architectural issues and low-level code quality problems.

Based on our goal, we formulate the following Research Questions (RQ_s)

RQ_1. Are SAWs and AS correlated?

Investigating the correlation between AS and SAW enables us to assess the relationship between high-level architectural issues and low-level code quality problems. We aim to determine whether ASs are correlated with SAWs and the strength of this correlation. To address RQ_1, we collected SAW and AS presence per software package. To assess statistical significance, we conjectured one **hypothesis** (H_1) as follows:

- H_{11}: *There is a statistically significant correlation between the presence of static analyzer warnings and the presence of architectural smells.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- H_{01}: *There is no statistically significant correlation between the presence of static analyzer warnings and the presence of architectural smells.*

We then tested whether the SAW influenced the presence of ASs computing Spearman's ρ correlation coefficient [41] between the total number of specific SAWs and ASs, averaged per software package (see Section 3.4).

Moreover, different static analyzer warnings can lead to specific architectural smells. Therefore, assessing whether a specific SAW's presence is a possible indicator of specific AS presence is essential. Therefore, we ask:

RQ_2. Do SAWs influence AS presence?

In the long run, architectural smell can silently deteriorate complex software architecture. Hence, we investigate the characteristics of SAW affecting software projects and leading to AS to identify common traits among projects prone to AS and assess further factors contributing to AS emergence and persistence. To address RQ_2 we analyzed the co-occurrence of SAW and ASs. Let σ be a SAW and α be an AS, and η be a software entity. We state that σ co-occur with α, if given η *such that* σ and α affect contemporary η. To assess statistically significance, we conjectured two **hypotheses** (H_2 and H_3) as follows:

- H_{12}: *There is a statistically significant difference in the co-occurrence of pairs of SAW and AS.*
- H_{13}: *There is a statistically significant difference in the co-occurrence of SAT-specific SAWs' and AS presence.*

Hence, we defined the **null hypotheses** (H_0) as follows:

- H_{02}: *There is no statistically significant difference in the co-occurrence of pairs of SAW and AS.*
- H_{03}: *There is no statistically significant difference in the co-occurrence of SAT-specific SAWs' and AS presence.*

We computed the co-occurrence of each SAW and AS. For each project package, we counted the number of healthy CDs, UDs, and HLs (and combinations). We tested H_2 and H_3 with Wilcoxon Paired Signed Ranked Test (WPT) (see Section 3.4).

Finally, assessing the correlation between SAW and AS and understanding how this relationship influences the presence of AS in software projects merely scratches the surface. Indeed, it falls short, considering practitioners' limited time for remediation efforts. Therefore, prioritizing these efforts becomes paramount. Thus, we ask:

RQ$_3$. Are SAWs effective for AS remediation effort prioritization?

Effort prioritization is essential to avoid wasting resources on noncritical tasks [42]. No prior study has investigated AS prioritization via SAW severity. According to the "no free lunch theorem", currently, there is no default severity for code quality checks universally accepted by our community [43]. Therefore, in our study, we investigate exploiting SAW characteristics, such as SonarQube rules severity and cost estimation, the prioritization effort for AS severity according to Arcelli Fontana et al. [5]. We referred to recent empirical studies to assess the accepted severity of AS [44, 45, 46]. More specifically, Sas and Avgeriou [46] propose, on average, an ordinal scale of one to ten, where one is the least severe AS and ten is the highest. Hence, we adopted the average value of CD (5), UD (7), and HL (9). Moreover, regarding SAW severity, we compiled a comprehensive dataset in our previous study [2] focused on SonarQube. Without the risk of generalizability issue [43], we focused this RQ on SonarQube-specific SAWs. To address RQ$_3$ we quantify the **probability of a SAW inducing an AS** ($\mathcal{P}$) as the relative frequency at which the particular SAW contributed to the occurrence of an AS. To assess statistical significance, we conjectured one **hypothesis** (H_4) as follows:

- H_{14}: *There is a statistically significant difference in the prioritized effort between the three rankers.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- H_{04}: *There is no statistically significant difference in the prioritized effort between the three rankers*

We computed $\mathcal{P}$ for each SAT, SAW, and AS. We ranked the SAWs based on their severity [2], $\mathcal{P}$, and AS severity [46]. We tested H_4 with WPT (see Section 3.4).

3.2. Study Context

We selected projects from the Qualitas Corpus (QCD) collection of software systems [47], using the compiled version of the QCD [22]. Figure 1 shows the word cloud composed by the names of the 112 java projects in the dataset crafted by Tempero et al. [22]. Table 2 details metrics on the QCD.

3.3. Study setup and data collection

This section presents our data collection methodology. Figure 2 presents the study workflow. Our data collection comprised three steps:

1. We analyzed the QCD with four SATs and averaged the results for project packages;
2. We separately analyzed QCD with the Arcan tool;

Figure 1: Qualitas Corpus WordCloud [22].

3. To get our study results, we analyzed the finalized dataset, comprising all SAW detected by the three SATs and the AS detected by the Arcan tool grouped by software package.

We analyzed the entire data set (112 Java projects) from the Qualitas Corpus [47]. However, we obtained results for only 103 projects because the computational time for the remaining ones would have been excessively long.

3.4. Data Analysis

We considered the classes affected by at least an AS and one static analysis warning at the class or package levels. One or more static analysis warnings could infect the same set of classes for each AS. This is because an AS may involve more than one class, while static analysis warnings involve only one class. In the case of projects not infected by static analysis warnings or AS, we did not consider them for the analysis.

To answer **RQ$_1$**, we collected the QCD projects and ran the SAT and ARCAN. We merged the outputs of the tool. In our context, the dependent variable is the AS, and the independent variables are the SAW. To compute the correlation coefficients, we need to assess the distributions of our variable. Therefore we conjectured one **hypothesis** ($H_{\mathcal{N}}$) as follows:

- $H_{1\mathcal{N}}$: *The data do not follow a normal distribution.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- $H_{0\mathcal{N}}$: *The data follow a normal distribution.*

We tested $H_{\mathcal{N}}$ with the Anderson-Darling (AD) test [48]. The AD test assesses whether data samples derive from a specific probability distribution, such as the normal distribution. AD measures the difference between the sample data and the expected values from the tested distribution. More specifically, it evaluates differences in the cumulative distribution function (CDF) between the observed data and the hypothesized distribution [48].

The AD test and the Shapiro-Wilk (SW) test [49] are both statistical tests used to assess the normality of data.

Metric	Value
Number of Projects	112
Lines of Code (LOC)	> 18 million
Packages	16,509
Classes Analyzed	202,052
Interfaces Analyzed	22,115
Methods Analyzed	464,893
Contexts Covered	IDEs, SDKs, databases, 3D/graphics/media, diagram/visualization libraries and tools, games, middlewares, parsers/generators/make tools, programming language compilers, testing libraries, and tools.
Available Metrics	LOC, NOP, NOCL, NOI, NOM, NOA, NORM, PAR, NSM, NSA. CK Metrics: WMC, DIT, NOC, LCOM HS. - MLOC, SIX, VG, NBD, RMD. - CA, CE, I, A.

Table 2: Summary of Qulitas Corpus Metrics

Figure 2: Data Collection Workflow

The SW test focuses on the correlation between the observed data and the expected values under a normal distribution, emphasizing the smallest and largest values in the dataset. Therefore, according to Mishra et al. [50], SW is more appropriate when targeting small datasets (≤ 50 samples). On the other hand, the AD test considers a wider range of values, including those in the middle of the distribution, providing a more sensitive evaluation of normality, especially for larger sample sizes than our own. Therefore, we preferred AD over SW [51]. AD is also considered one of the most powerful statistical tools for detecting most deviations from normality [51, 52].

Test results allowed us to **reject the null hypothesis** for each test case, $H_{0\mathcal{N}}$, asserting that the **data was not normally distributed**. Therefore, we chose Spearman's ρ [41] to compute the correlation in place of Pearsons, which requires a normal distribution [53]. The non-parametric Spearman's rank correlation coefficient, denoted by ρ, measures statistical dependence between two variables without necessarily assuming a linear relationship or even normality of those variables. Spearman's Rho measures the strength and direction of the monotonic relationship between the ranks of paired data points. Table 3 presents ρ values interpretation according to Dancey and Reidy [54].

To answer $\mathbf{RQ_2}$, we computed Spearman's ρ for each SAW-AS combination. We used the interquartile ratio (IQR) method [55] to select the top correlated SAW for each SAT and AS, i.e., we computed the quartiles. We selected only the SAW with their Spearman's ρ value higher or equal to the third quartile (Q3), thus selecting the top 25%. We computed the $\mathcal{P}$ as follows: Let Σ_{SAT_i} represent the set of Static Analysis Warnings specific to a particular Static Analysis Tool SAT_i. Then, $\sigma \in \Sigma_{\text{SAT}_i}$ signifies that σ belongs to the set of SAWs specific to SAT_i; thus σ_j is a specific SAW and $\alpha \in AS$ represents a specific AS.. Therefore, we define the relative frequency $\mathcal{P}(\sigma_j, \alpha)$ as:

$$\mathcal{P}(\sigma_j, \alpha) = \frac{|\sigma_j \text{ in } \alpha|}{|\sigma \text{ in } \alpha|}$$

Where:

- The numerator represents the number of times the specific SAW, $\sigma_j \in \Sigma_{\text{SAT}_i}$, appears within the given AS, α.
- The denominator represents the total count of all SAWs belonging to a specific SAT, σ, in the given AS, α.

More specifically, we computed $\mathcal{P}$ per each possible combination of AS. Therefore our data computed a $\mathcal{P}$-value for the 3 ASs, the healthy instances, and for combinations of ASs. We tested H_2, and H_3 via the WPT [56], which is a non-parametric statistical test that compares two related samples or paired data. WPT uses the absolute difference between the two observations to classify and then compare the sum of the positive and negative differences. The test statistic is the lowest of both. We selected WPT to test H_2 and H_3 because the data was not normally distributed; hence, we used it instead of the paired t-test, which assumes a normal data distribution.

Moreover, due to the high number of statistical tests performed, we needed to apply p-value correction. Correcting p-values is essential in multiple hypothesis testing

Interpretation	Perfect	Strong			Moderate			Weak			Zero
Correlation	+ 1	+ 0.9	+0.8	+ 0.7	+0.6	+ 0.5	+ 0.4	+ 0.3	+ 0.2	+ 0.1	0
	- 1	- 0.9	- 0.8	- 0.7	- 0.6	- 0.5	- 0.4	- 0.3	- 0.2	- 0.1	0

Table 3: Spearman's ρ interpretation (RQ_1)

to control the increased risk of Type I errors. When multiple tests are conducted simultaneously, the probability of incorrectly rejecting at least one true null hypothesis increases. A classical p-value correction technique is the Bonferroni correction. However, Bonferroni is known to offer a conservative correction [57]. Therefore, we decided to use the Benjamini & Hochberg (BH) p-value correction [58], which uses the false discovery rate (FDR) when performing multiple hypothesis tests as follows [59]:

1. **Rank the p-values**: Sort the p-values obtained from multiple tests in ascending order, denoted as $p_{(1)} \leq p_{(2)} \ldots \leq p_{(m)}$ for their respectively defined hypotheses $H_{(1)}, H_{(2)}, \ldots, H_{(m)}$.

2. **Assign ranks**: Assign each p-value a rank, i, with $p_{(1)}$ being the smallest p-value and $p_{(m)}$ being the largest.

3. **Calculate the corrected p-values**: For each p-value $p_{(i)}$, calculate the BH-corrected p-value using the formula:

$$q_{(i)} = \min_{j \in \{1,\ldots,m\}} \left\{ \frac{m \cdot p_{(j)}}{j}, 1 \right\}$$

   where m is the total number of tests, $q_{(i)}$ is the corrected probability for the hypothesis at moment i and $p_{(j)}$ given the order rank $j \in \{1, \ldots, m\}$ of p-values, which stands as the probability at the rank position j.

4. **Determine correction index**: The detailed procedure identified the largest index k for which BH holds and rejects null hypotheses as follows:

$$k = \max \left\{ i \in 1, \ldots, m \mid p_{(i)} \leq i \frac{\alpha}{m} \right\}$$

   which is automatically computed with existing analytical software.

5. **Corrected hypothesis testing**: Compare the corrected p-values $q_{(i)}$ from each hypothesis $H_{(1)}, H_{(2)}, \ldots, H_{(m)}$ against the k index, that is:

$$p_{(i)} \leq i \frac{\alpha}{m}$$

   to the desired critical value, α (e.g., 0.05), to determine which hypotheses to reject.

Since we performed the BH correction on the p-values of the defined hypotheses, we maintained the standard value of the critical value α on 0.05, hence gaining power on controlling for error Type I and Type II [59].

To answer $\mathbf{RQ_3}$, we leverage our definition of $\mathcal{P}$, the default severity associated with the specific SAW, and the severity associated with the specific AS to build three different rankers. The ranking idea stemmed from previous research by Çarka et al. [31], in which PofB and PopT metrics were normalized and used to evaluate classifier prediction concerning inspection effort (see Section 2.2).

Following the definition of PofB and Popt, we designed three rankers. More specifically, we ordered the data according to:

- SAW severity: we used SonarQube severity associated with each SAW.
- $\mathcal{P}$: we used $\mathcal{P}$ and created separate rankings one for each AS combination.
- AS severity: we used the known AS severity to order the data.

We note that the AS severity-based order is a special case; by design, that ranker is the *optimal* one because it uses the AS severity. RQ_3 aims to use the other two rankers to prioritize those SAWs that are more prone to induce ASs. Therefore, the optima ranker is needed to compare the results of the other two. According to the interpretation of PofB and PopT, we selected the first x% of the data from the three rankings. Therefore, for each selected percentage of data, we computed how many Critical, High, and Medium AS-prone SAWs we would capture.

3.5. Replicability

We provide a replication package containing the raw data, including the instructions for the SAT execution, the list of projects infected by SAW and AS, and the full statistical test results for both the normality hypothesis and for the RQ's hypotheses [5].

4. Results

In this section, we reported the results we obtained to answer our RQs.

[5] https://zenodo.org/doi/10.5281/zenodo.11366846

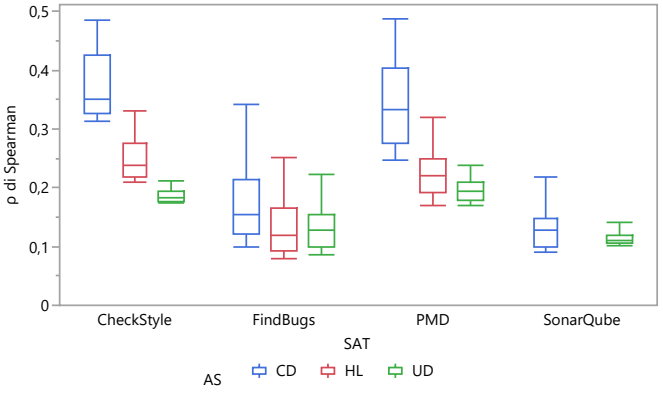

Figure 3: Spearman's ρ average distribution for SAW by AS and SAT (RQ_1)

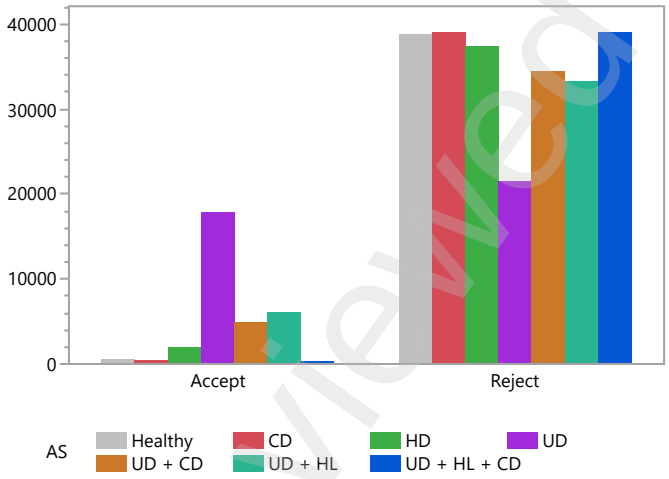

Figure 4: Distribution of H_2 test results (RQ_2)

4.1. Static Analysis Warnings and Architectural Smell Correlation (RQ_1)

We tested H_1 with Spearman's ρ. We excluded all cross-correlations among SAWs and focused on the correlations between SAW and AS. Figure 3 presents the distribution of Spearman's ρ averaged for SAW by AS and SAT. We can reject the null hypothesis for each of the 663 cases. Hence, we can affirm that **there is a statistically significant correlation between the presence of static analyzer warnings and the presence of architectural smells**. More specifically, SAWs were weak to moderately correlated to AS for each AS across SAT.

4.2. Static Analysis Warnings inducing Architectural Smell (RQ_2)

We tested H_2 with WPT. We rejected the null hypothesis for 243,258 out of 275,380 cases, i.e., 88%. Therefore, we can, on average, affirm that **there is a statistically significant difference in the co-occurrence of pairs of SAW and AS.** More specifically, Figure 4 presents the distribution of H_2 test results. It is worth noticing that in the case of UD, there was almost an equal probability of rejecting or accepting the null hypothesis across SAWs. Moreover, we note that there is an equal probability of rejecting the null hypothesis in the case of Healthy cases as well as those affected by all of the three ASs.

Furthermore, we tested H_3 with WPT. We can reject the null hypothesis in 30 out of 42 cases, i.e., 72% of the cases. Table 4 shows the test cases for which we could not reject the null hypothesis. Therefore, we can affirm that **there is a statistically significant difference in the co-occurrence of SAT-specific SAWs and ASs presence**.

More specifically, Table 5 presents the top five SAW co-occurring with specific AS. We selected the top five SAWs that co-occur the most for each AS, including the "healthy carriers". We note that the most co-occurring SAWs belong to FindBugs. Similarly, FindBugs also has the most "healthy carriers" co-occurring SAWs. Moreover, where three ASs affect a single package, the FindBugs SAW co-occurs each time. Futhermore, the least co-occurring AS is UD.

4.3. Architectural Smell Prioritization (RQ_3)

Figure 5 presents the distribution of $\mathcal{P}$ averaged among SAW across SAT per AS. Therefore, on average, Figure 5 presents the percentage of occurrences of SAWs related to the occurrence of one or more ASs or their absence (i.e., healthy). On average, SAWs belonging to SonarQube are more prone to raise warnings on healthy packages and classes. Most SAWs co-occur with CD AS and have a similar presence among the SATs. Moreover, second to CD, SAWs co-occurred with all three ASs. In this context, FindBugs' SAWs exhibit an average higher co-occurrence.

Figure 6 presents the SAW prioritization based on the three rankings, i.e., SAW Severity-based, $\mathcal{P}$-based, and optimal. We computed how many medium, high, or critical AS could be discovered by inspecting the first x% of data ranked by one of the three methods. According to its design, the optimal ranking prioritized the SAW co-occurring with high and critical AS. The SAW severity-based and UD+CD rankers demonstrate the most balanced performance, while the ranking based on SAW co-occurring with CD favors moderately severe AS.

Finally, we tested H_4 with the WPT. We could reject the null hypothesis in all cases for high and critical AS severities, in which the optimal ranker was compared with the SAW-severity-based and the $\mathcal{P}$ based rankers. Conversely, we could reject the hypothesis when comparing the SAW-severity-based and the $\mathcal{P}$ based rankers but only the specific case when comparing the UD + HL + CD proneness based on $\mathcal{P}$. Futhermore, when the difference was statically significant, the gains were in favor of the optimal rankers, as expected.

We included the full table and the graphical representation of the gains among the approaches in the replication package for space constraint.

Table 4: Wilcoxon All Pair Test, i.e., H_{03} p-value (RQ_2)

AS	**Healthy**	**CD**	**CD**	**CD**	**CD**	**CD**
SAT	PMD	PMD	FindBugs	PMD	SonarQube	SonarQube
Compared with	CheckStyle	CheckStyle	CheckStyle	FindBugs	CheckStyle	FindBugs
p-value	0,76	0,25	0,44	0,85	0,58	0,20

AS	**CD**	**UD**	**UD + CD**	**HL**	**UD + HL + CD**	**UD + HL + CD**
SAT	SonarQube	FindBugs	PMD	PMD	FindBugs	PMD
Compared with	PMD	CheckStyle	CheckStyle	CheckStyle	CheckStyle	CheckStyle
p-value	0,02	1,00	0,14	0,40	0,03	0,28

Table 5: The top five SAW co-occurring with specific AS

SAT	**SAW**	**AS**	**% co-occurring**
FindBugs	AT_OPERATION_SEQUENCE_ON_CONCURRENT_ABSTRACTION		100
FindBugs	DMI_EMPTY_DB_PASSWORD		100
FindBugs	DMI_ENTRY_SETS_MAY_REUSE_ENTRY_OBJECTS	UD + HL + CD	100
FindBugs	DMI_RANDOM_USED_ONLY_ONCE		100
FindBugs	EQ_COMPARING_CLASS_NAMES		100
PMD	UseStringBufferForStringAppends		6,132
PMD	AssignmentInOperand		4,856
PMD	code_smells:spaghetti_code	UD + HL	3,448
PMD	UseCollectionIsEmpty		2,435
PMD	BooleanGetMethodName		1,904
SonarQube	squid:ClassCyclomaticComplexity		3,418
PMD	AvoidPrefixingMethodParameters		3,222
SonarQube	squid:S1197	UD + CD	2,300
FindBugs	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE		2,270
FindBugs	BC_UNCONFIRMED_CAST		2,178
SonarQube	S2386		0,429
SonarQube	S1444		0,294
SonarQube	S1141	UD	0,267
SonarQube	S1118		0,242
SonarQube	S1226		0,223
FindBugs	DMI_COLLECTION_OF_URLS		40,000
FindBugs	OBL_UNSATISFIED_OBLIGATION		33,871
CheckStyle	NoWhitespaceAfterCheck	HL	31,748
FindBugs	EQ_DOESNT_OVERRIDE_EQUALS		21,739
PMD	LongVariable		18,457
FindBugs	NS_NON_SHORT_CIRCUIT		88,235
FindBugs	ICAST_INTEGER_MULTIPLY_CAST_TO_LONG		87,500
FindBugs	RV_CHECK_FOR_POSITIVE_INDEXOF	CD	80,000
FindBugs	DM_GC		78,571
FindBugs	VA_FORMAT_STRING_USES_NEWLINE		75,000
FindBugs	NM_METHOD_NAMING_CONVENTION		89,865
FindBugs	RI_REDUNDANT_INTERFACES		86,029
FindBugs	SIO_SUPERFLUOUS_INSTANCEOF	Healthy	70,130
FindBugs	EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS		68,421
PMD	CompareObjectsWithEquals		66,212

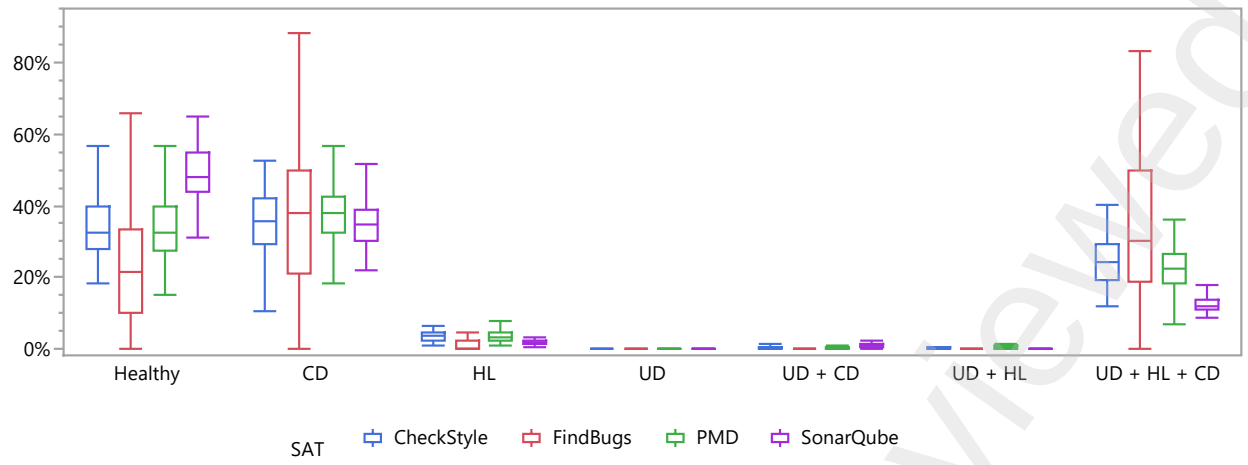

Figure 5: Distribution of $\mathcal{P}$ averaged among SAW across SAT per AS (RQ_2)

Moreover, we can not reject the null hypothesis for the medium risk for each ranker. Therefore, the optimal ranker behaves similarly to the other two rankers.

Hence, we can affirm that **there is a statistically significant difference in the prioritized effort between the three rankers**.

Nevertheless, based on the results, we can affirm that **SAW prioritization based on $\mathcal{P}$ can effectively guide AS remediation effort**.

5. Discussion

This section discusses the results and presents lessons learned based on them.

Our findings reveal a moderate correlation between SAWs and ASs. However, different combinations of SATs and SAWs significantly influence the occurrence of ASs. Thus, our study's key takeaway is that SAWs have a statistically discernible influence on ASs, despite the moderate correlation.

Furthermore, specific ASs show varying degrees of susceptibility to SAWs. For instance, as illustrated in Figure 4, UD's acceptance or rejection of H_{03} was almost equally probable. Likewise, SAWs that consistently fail to trigger ASs, i.e., Healthy, consistently reject the null hypothesis. As a result, this **SAWs, acting as "healthy carriers", can be safely discarded** when prioritizing AS remediation efforts.

More specifically, the prevalence of FindBugs SAWs among the top co-occurring SAWs indicates that FindBugs might be more effective at finding issues related to ASs. This shows that FindBugs is effective at finding certain AS patterns. Thus, future research efforts should focus on the difference between FindBugs SAW and the other SATs SAWs. Thus grasping the characteristics that make FindBugs SAWs co-occur with all three ASs and with the "healthy carriers".

Futhermore, future research efforts should focus on identifying UD AS, being the least co-occurring AS may suggest that these issues are not sufficiently characterized by the considered SAWs and SATs.

Therefore, by avoiding wasting remediation time on healthy carries, regarding RQ_3, our study highlighted that prioritizing AS remediation efforts based on SAW severity, or SAW proneness to specific AS with a limited inspection window, results in similar rankings to the optimal one. Healthy carriers represent 33.79% of our dataset. Hence we can **safely drop a third of the SAW**, focusing on the most AS-prone to effectively address AS without prior AS-severity or discovery knowledge.

6. Threats to Validity

In this section, we discuss the threats to the validity of our study. We categorized the threats into Construct, Internal, External, and Conclusion, following the guidelines defined by Wohlin et al. [39].

Construct Validity. Construct validity concerns how our measurements reflect what we claim to measure [39].

Our specific design choices may impact our results, including our measurement process and data filtering. To address this threat, we based our choice on past studies [5, 22, 60, 46] and used well-established guidelines in designing our methodology [61, 40]. More specifically, we used four static analysis tools considered to be the most adopted by developers to detect software quality warnings [13] and ARCAN to detect the four architectural smells.

Nevertheless, we acknowledge that the analyzed SAW does not represent all the possible SAWs currently available in the state-of-the-art. We mitigated this threat by selecting the most popular SAT. Nonetheless, future works will focus on extending the current selection of SAW. Similarly, we analyzed only three AS due to the lack of AS detection tools. Future works will focus on expanding the AS detectable by ARCAN to mitigate this threat.

Internal Validity. Internal Validity is the extent to which an experimental design accurately identifies a cause-and-effect relationship between variables [39]. Our study

Figure 6: Comparison of Inspection Effort Analysis based on SAW Severity, $\mathcal{P}$, and AS Severity.

relies on a large-scale analysis of 103 popular Java projects in the Qualitas Corpus dataset. However, they can potentially be biased from their project selection. We address this threat by designing our inclusion criteria based on past studies [60, 47, 22].

External Validity. External validity concerns how the research elements (subjects, artifacts, etc.) represent actual elements [39]. External validity concerns how the research elements (subjects, artifacts, etc.) represent actual elements [39]. The analyzed project comes from the GitHub public repository. Mining versioning systems, particularly GitHub, also threaten external validity. More specifically, GitHub's user base predominantly comprises developers and contributors to open-source projects, potentially skewing findings towards this specific demographic. In our context, this is not a real threat because it is the study's focus. Moreover, the dynamic nature of GitHub, with frequent updates, forks, and merges, poses challenges in ensuring the stability and consistency of data over time. We addressed this issue by providing the raw data in our replication package. Furthermore, the accessibility of GitHub data is subject to various permissions and restrictions set by project owners, potentially hindering reproducibility and transparency in research. We addressed this research, limiting our data collection to public repositories. Nonetheless, the vast diversity of projects analyzed in our study presents a significant diversity in terms of programming languages, project sizes, and development methodologies, thus aiding the generalizability of results.

Conclusion Validity focuses on how we draw conclusions based on the design of the case study, methodology, and observed results [39]. The distribution of our data determines the type of statistical tools we can use to test our hypothesis. We tested the data to asses if it departed from normality with the AD test, and we could reject the null hypothesis for each test. Therefore we selected the WT test instead of the t-test as Spearman's ρ in place of Pearsons. Furthermore, statistical tests threaten the conclusion's validity regarding the appropriateness of statistical tests and procedures, such as assumption violation, multiple comparisons, and Type I or Type II errors. We address this issue using WT instead of the t-test due to the rejection of the normal data distributions. Moreover, we applied the HB correction [62] to balance type 1 and type 2 errors well. We also set our alpha to 0.01, hence reducing it from the standard value of 0.05 due to the many statistical tests we performed.

Spearmans ρ is a non-parametric measure of the strength and direction of a correlation between two variables, but due to multiple factors, this approach may hinder the conclusions validity. For instance, the correlation coefficient is unreliable with a small sample size and cannot accurately represent the relationship between variables. Moreover, Spearman assumes a linear monotonic correlation between the variables; if the assumption were to prove invalid or not applicable, Spearman's correlation might not accurately measure the strength and direction of the association. We have mitigated this issue by ensuring a representative sample and checking the data distribution. We could mitigate this issue by using other non-parametric measures of correlation. Still, our analysis found that Spearman's ρ was more suitable than other correlation measures, such as Kendall's τ, because it can effectively handle tied observations in the data. On the other hand, Kendalls τ relies on concordant and discordant pairs of observations.

7. Related Work

This section discusses works related to our study regarding SATs and AS.

7.1. Static Analysis Tools

In this section, we report the relevant work on SAT focusing on their usage, warnings, and the detected issues [63, 64, 65]. SATs increasing in popularity [11, 66] given their ease-to-use [24]. Recently researchers focus on analyzing SAT usage [67, 68, 69, 70] and effectiveness [13]. For instance, SATs effectively improve bug prediction models [71]. Developers use SAT to inspect software codebases, identify bugs [72, 73], evaluate code quality [74, 75, 76], and address coding issues [77, 78, 15].

Recent studies has explored SAT's application across various programming languages [65], their configurations [65], and their evolution within software systems [79, 80, 77]. However, fewer studies have focused on the impact of SAT on software quality [79, 81, 15, 82, 83].

Moreover, the research community also focused on prioritization strategies for addressing warnings. For instance, researchers proposed prioritization based on removal times [84], or developer engagement in remediating specific warnings [76]. Moreover, studies have compared estimated warning resolution times with actual developer efforts [80, 85, 86, 87].

Researchers have extensively studied SonarQube regarding the impact of SAW on software quality. They investigated the fault-proneness of SonarQube rules [79, 15, 88] and their susceptibility to changes [81, 15]. Findings indicate that SonarQube rules increase change-proneness at the class level, while correlations between faults and rules can reduce fault-proneness [79, 15].

Similarly, researchers have examined PMD warnings concerning their impact on pull request acceptance. Results show quality warnings do not affect pull request acceptance [82]. A comprehensive study [83] on warning trends in OSS projects using PMD highlighted that large-scale changes in SAT warnings often result from coding style modifications, such as braces and naming conventions. The presence of PMD in build configurations positively impacts defect density, suggesting an improvement in external quality.

Recently, Wright et al. [89] investigated why developers use SAT, revealing new tool requirements such as systems for recommending warnings based on developer expertise and collaborative interfaces for warning analysis. Conversely, Johnson et al. [74] examined why developers do

not widely use SAT tools, identifying false positives and the presentation of warnings as significant barriers. Furthermore, Yang et al. [90] explored enhancing the utility of static code warning tools through data mining algorithms that filter out commonly ignored warnings, using FindBugs as a case study.

7.2. Architectural Smells

AS have been less thoroughly investigated than code smells. While numerous studies have explored correlations between code smells, research into the relationship between code smells and AS has been limited. One study found a minimal correlation between AS and code smells, suggesting that AS cannot be reliably inferred from code smells [19].

The connection between code anomalies, similar to code smells, and AS has also been studied with mixed results. Some findings indicated that many detected code anomalies did not correspond to architectural issues [17], while other results showed that over 80% of architectural problems were linked to code smells [18]. Specific code smells like Long Method, and God Class were consistently associated with architectural problems.

Oizumi et al. [91] investigate the aggregation of code anomalies and their relationship to AS and show that a single anomaly can signal an architectural problem. Specifically, 50% of syntactic aggregations and 80% of semantic aggregations are linked to design issues [92, 91]. A study does not find a strong correlation between AS and architectural degradation, outlining that AS cannot explain architectural degradation alone [93], while Sas et al. [94] highlights the effects of AS on long-term system maintainability and evolvability, supported by interview excerpts. Practitioners highlight which parts of the AS analysis offer actionable insights for planning refactoring activities.

Moreover, Mumtaz et al. [95] shows that community smells, particularly the Missing Links smell, are related to design smells. They discuss specific refactoring techniques that concurrently address community and design smells, managing social and technical issues together.

Finally, Fontana et al. [19] on the correlation between technical debt indices based on code-level issues from SonarQube revealed no correlation, suggesting differing impacts based on the features considered [26]. Although several tools have been developed for detecting architectural debt, including DV8, Designite, Jarchitect, and others, which focus on AS in various programming languages [96, 97, 98], no prior studies have examined correlations between AS and SAW as we have addressed in this paper.

8. Conclusion

Our results show a statistically significant positive correlation between SAWs and AS, showing weak to moderate correlations across different tools and SAWs and AS in 661 out of 663 tests. In terms of co-occurrence, we found a statistically significant difference in the pairs of SAWs and ASs, with 88% of the Wilcoxon All-Pair Ranking Test cases rejecting the null hypothesis. Specifically, we showed that SAWs can influence the presence of ASs. When prioritizing architectural smells, our findings indicate that using a prioritization method based on the probability of SAW occurrences ($\mathcal{P}$) can effectively guide remediation efforts, revealing a statistically significant difference between the three ranking methods tested. Our proposed approach to SAW prioritization highlights that, in the absence of AS indication, SAW severity and our empirically computed $\mathcal{P}$ together with the possibility of dropping a third of the data to inspect, proved to be effective ASs remediation effort prioritization.

Future research efforts should enhance SAT capabilities to detect the most AS-prone SAWs more accurately, improving AS remediation prioritization. Additionally, research should aim to deepen the analysis of SAWs' AS-proneness to better trace AS back to SAWs. This will help understand the motivations or root causes linking specific SAWs to specific AS.

References

[1] M. Esposito, D. Falessi, Uncovering the hidden risks: The importance of predicting bugginess in untouched methods, in: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023, pp. 277–282. doi:10.1109/SCAM59687.2023.00039.

[2] V. Lenarduzzi, N. Saarimäki, D. Taibi, The technical debt dataset, in: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'19, Association for Computing Machinery, New York, NY, USA, 2019, p. 211. URL: https://doi.org/10.1145/3345629.3345630. doi:10.1145/3345629.3345630.

[3] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223–235. URL: http://www.sciencedirect.com/science/article/pii/S0950584916304128. doi:https://doi.org/10.1016/j.infsof.2017.08.008.

[4] M. Fowler, K. Back, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, E. D. Nitto, Arcan: A tool for architectural smells detection, in: International Conference on Software Architecture Workshops, ICSA, 2017, pp. 282–285.

[6] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: CSMR 2009, 2009, pp. 255–258.

[7] F. Arcelli Fontana, R. Roveda, M. Zanoni, Tool support for evaluating architectural debt of an existing system: An experience report, in: Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016), ACM, Pisa, Italy, 2016. To appear.

[8] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, I. Gorton, Measure it? Manage it? Ignore it? Software practitioners and technical debt, Symposium on the Foundations of Software Engineering (2015) 50–60.

[9] R. Nord, I. Ozkaya, P. Kruchten, M. Gonzalez-Rojas, In search of a metric for managing architectural technical debt, in: European Conference on Software Architecture (ECSA), 2012, pp. 91–100.

[10] P. Rachow, 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.

[11] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019) 1–39.
[12] P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, A. Tsintzira, An overview and comparison of technical debt measurement tools, IEEE Software (2021).
[13] M. Esposito, V. Falaschi, D. Falessi, An extensive comparison of static application security testing tools, arXiv preprint arXiv:2403.09219 (2024).
[14] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, R. Oliveto, Smells like teen spirit: Improving bug prediction performance using the intensity of code smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 244–255. doi:10.1109/ICSME.2016.27.
[15] V. Lenarduzzi, N. Saarimki, D. Taibi, Some sonarqube issues have a significant but smalleffect on faults and changes. a large-scale empirical study, Journal of Systems and Software 170 (2020).
[16] F. Arcelli Fontana, M. Zanoni, On investigating code smells correlations, in: International Conference on Software Testing, Verification and Validation Workshops (ICSTW), RefTest Workshop, Berlin, Germany, 2011, pp. 474–475.
[17] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, A. von Staa, Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems, in: International Conference on Aspect-oriented Software Development (AOSD '12), 2012, pp. 167–178.
[18] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, A. von Staa, On the relevance of code anomalies for identifying architecture degradation symptoms, in: Conference on Software Maintenance and Reengineering (CSMR 2012), 2012, pp. 277–286.
[19] F. A. Fontana, V. Lenarduzzi, R. Roveda, D. Taibi, Are architectural smells independent from code smells? an empirical study, Journal of Systems and Software 154 (2019) 139–156.
[20] A. B. Desai, J. K. Parmar, Refactoring cost estimation (rce) model for object oriented system, in: 2016 IEEE 6th International Conference on Advanced Computing (IACC), 2016, pp. 214–218. doi:10.1109/IACC.2016.48.
[21] S. Deeb, M. BenIdris, H. Ammar, D. Dzielski, Refactoring cost estimation for architectural technical debt, International Journal of Software Engineering and Knowledge Engineering 31 (2021) 269–288. URL: https://doi.org/10.1142/S021819402150008X. doi:10.1142/S021819402150008X.
[22] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, APSEC 2010 (2010) 336–345. doi:10.1109/APSEC.2010.46.
[23] F. A. Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, 2016, pp. 433–437. URL: https://doi.org/10.1109/ICSME.2016.33. doi:10.1109/ICSME.2016.33.
[24] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, M. Di Penta, How open source projects use static code analysis tools in continuous integration pipelines, in: Int. Conf. on Mining Software Repositories, 2017, pp. 334–344.
[25] R. C. Martin, Object oriented design quality metrics: An analysis of dependencies, ROAD 2 (1995).
[26] R. Roveda, F. A. Fontana, I. Pigazzini, M. Zanoni, Towards an architectural debt index, in: 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018), 2018, pp. 408–416.
[27] A. Martini, F. Arcelli Fontana, A. Biaggi, R. Roveda, Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company: 12th European Conference on Software Architecture,, 2018, pp. 320–335.
[28] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2007.
[29] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for Software Design Smells, 1 ed., Morgan Kaufmann, 2015.
[30] H. A. Al-Mutawa, J. Dietrich, S. Marsland, C. McCartin, On the shape of circular dependencies in java programs, in: ASWEC 2014, 2014, pp. 48–57.
[31] J. Çarka, M. Esposito, D. Falessi, On effort-aware metrics for defect prediction, Empirical Software Engineering 27 (2022) 152.
[32] H. Chen, W. Liu, D. Gao, X. Peng, W. Zhao, Personalized defect prediction for individual source files, 44 (2017) 90–95. URL: https://doi.org/10.11896/j.issn.1002-137X.2017.04.020. doi:10.11896/j.issn.1002-137X.2017.04.020.
[33] S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, IEEE Trans. Software Eng. 46 (2020) 1267–1293. URL: https://doi.org/10.1109/TSE.2018.2877612. doi:10.1109/TSE.2018.2877612.
[34] X. Xia, D. Lo, S. J. Pan, N. Nagappan, X. Wang, HYDRA: massively compositional model for cross-project defect prediction, IEEE Trans. Software Eng. 42 (2016) 977–998. URL: https://doi.org/10.1109/TSE.2016.2543218. doi:10.1109/TSE.2016.2543218.
[35] H. Tu, Z. Yu, T. Menzies, Better data labelling with emblem (and how that impacts defect prediction), IEEE Transactions on Software Engineering (2020) 1–1. doi:10.1109/TSE.2020.2986415.
[36] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: T. J. Ostrand (Ed.), Proceedings of the 5th International Workshop on Predictive Models in Software Engineering, PROMISE 2009, Vancouver, BC, Canada, May 18-19, 2009, ACM, 2009, p. 7. URL: https://doi.org/10.1145/1540438.1540448. doi:10.1145/1540438.1540448.
[37] E. Arisholm, L. C. Briand, M. Fuglerud, Data mining techniques for building fault-proneness models in telecom java software, in: ISSRE 2007, The 18th IEEE International Symposium on Software Reliability, Trollhättan, Sweden, 5-9 November 2007, IEEE Computer Society, 2007, pp. 215–224. URL: https://doi.org/10.1109/ISSRE.2007.22. doi:10.1109/ISSRE.2007.22.
[38] T. Yu, W. Wen, X. Han, J. H. Hayes, Conpredictor: Concurrency defect prediction in real-world applications, IEEE Trans. Software Eng. 45 (2019) 558–575.
[39] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012.
[40] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering (1994).
[41] C. Spearman, The proof and measurement of association between two things, The American Journal of Psychology 15 (1904) 72–101.
[42] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Transactions on software engineering 33 (2006) 33–53.
[43] M. Esposito, S. Moreschini, V. Lenarduzzi, D. Hästbacka, D. Falessi, Can we trust the default vulnerabilities severity?, in: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2023, pp. 265–270.
[44] F. A. Fontana, F. Locatelli, I. Pigazzini, P. Mereghetti, An architectural smell evaluation in an industrial context, ICSEA 2020 (2020) 68–74.
[45] I. Pigazzini, D. Foppiani, F. A. Fontana, Two different facets of architectural smells criticality: An empirical study., in: ECSA (Companion), 2021.
[46] D. Sas, P. Avgeriou, An architectural technical debt index based on machine learning and architectural smells, IEEE Transactions on Software Engineering (2023).
[47] R. M. Terra, L. F. Miranda, M. T. Valente, R. da Silva Bigonha, Qualitas.class corpus: a compiled version of the qualitas corpus, ACM SIGSOFT Software Engineering Notes 38 (2013) 1–4.
[48] T. W. Anderson, D. A. Darling, Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes, The Annals of Mathematical Statistics 23 (1952)

193 – 212. URL: https://doi.org/10.1214/aoms/1177729437. doi:10.1214/aoms/1177729437.
[49] S. Shaphiro, M. Wilk, An analysis of variance test for normality, Biometrika 52 (1965) 591–611.
[50] P. Mishra, C. M. Pandey, U. Singh, A. Gupta, C. Sahu, A. Keshri, Descriptive statistics and normality tests for statistical data, Annals of cardiac anaesthesia 22 (2019) 67–72.
[51] M. A. Stephens, Edf statistics for goodness of fit and some comparisons, Journal of the American statistical Association 69 (1974) 730–737. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1974.10480196. doi:10.1080/01621459.1974.10480196.
[52] M. A. Stephens, Tests based on edf statistics, in: Goodness-of-fit-techniques, Routledge, 2017, pp. 97–194.
[53] K. Pearson, Notes on regression and inheritance in the case of two parents proceedings of the royal society of london, 58, 240-242, K Pearson (1895).
[54] C. P. Dancey, J. Reidy, Statistics without maths for psychology, Pearson education, 2007.
[55] T. Hojo, K. Pearson, Distribution of the median, quartiles and interquartile distance in samples from a normal population, Biometrika (1931) 315–363.
[56] W. Conover, Practical nonparametric statistics, 3. ed ed., New York, NY [u.a.], 1999.
[57] Y. Hochberg, A sharper bonferroni procedure for multiple tests of significance, Biometrika 75 (1988) 800–802.
[58] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach to multiple testing, Journal of the Royal statistical society: series B (Methodological) 57 (1995) 289–300.
[59] F. Emmert-Streib, M. Dehmer, Large-scale simultaneous inference with hypothesis testing: multiple testing procedures in practice, Machine Learning and Knowledge Extraction 1 (2019) 653–683.
[60] M. Dilhara, A. Ketkar, D. Dig, Understanding software-2.0: A study of machine learning library usage and evolution 30 (2021).
[61] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Engg. 14 (2009) 131–164.
[62] J. H. Kim, I. Choi, Choosing the level of significance: A decision-theoretic approach, Abacus 57 (2021) 27–71. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/abac.12172. doi:https://doi.org/10.1111/abac.12172.
[63] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for java, in: Conference on Programming Language Design and Implementation, 2002, p. 234245.
[64] S. Heckman, L. Williams, A systematic literature review of actionable alert identification techniques for automated static code analysis, Information and Software Technology 53 (2011) 363–387. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.
[65] M. Beller, R. Bholanath, S. McIntosh, A. Zaidman, Analyzing the state of static analysis: A large-scale evaluation in open source software, in: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, 2016, pp. 470–481.
[66] V. Lenarduzzi, A. Sillitti, D. Taibi, A survey on code analysis tools for software maintenance prediction, in: Software Engineering for Defence Applications - SEDA 2018, volume 925 of *Advances in Intelligent Systems and Computing (AISC)*, Springer-Verlag, 2019.
[67] S. Wagner, J. Jürjens, C. Koller, P. Trischberger, Comparing bug finding tools with reviews and tests, in: International Conference on Testing of Communicating Systems, 2005, p. 4055.
[68] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, in: 27th International Conference on Software Engineering (ICSE), 2005, pp. 580–586.
[69] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, M. A. Vouk, On the value of static analysis for fault detection in software, IEEE Transactions on Software Engineering 32 (2006) 240–253.
[70] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, P. Balachandran, Making defect-finding tools work for you, in: 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, 2010, p. 99108.
[71] L.-P. Querel, P. C. Rigby, Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings, in: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, p. 892895.
[72] N. Rutar, C. B. Almazan, J. S. Foster, A comparison of bug finding tools for java, in: Symposium on Software Reliability Engineering, 2004, pp. 245–256.
[73] P. Tomas, M. J. Escalona, M. Mejias, Open source tools for measuring the Internal Quality of Java software products. A survey, Computer Standards and Interfaces 36 (2013) 244–255.
[74] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 672–681.
[75] M. Schnappinger, M. H. Osman, A. Pretschner, A. Fietzke, Learning a classifier for prediction of maintainability based on static analysis tools, in: 27th International Conference on Program Comprehension, 2019, p. 243248.
[76] D. Marcilio, R. Bonifcio, E. Monteiro, E. Canedo, W. Luz, G. Pinto, Are static analysis violations really fixed? a closer look at realistic usage of sonarqube, in: 27th International Conference on Program Comprehension (ICPC), 2019, pp. 209–219.
[77] N. Saarimäki, V. Lenarduzzi, D. Taibi, On the diffuseness of code technical debt in open source projects, in: International Conference on Technical Debt (TechDebt 2019), 2019.
[78] V. Lenarduzzi, A. Martini, D. Taibi, D. A. Tamburri, Towards surgically-precise technical debt estimation: Early results and research roadmap, in: International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2019, 2019, pp. 37–42.
[79] D. Falessi, B. Russo, K. Mullen, What if i had no smells?, International Symposium on Empirical Software Engineering and Measurement (ESEM) (2017) 78–84.
[80] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, A. Ampatzoglou, How do developers fix issues and pay back technical debt in the apache ecosystem?, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 153–163.
[81] I. Tollin, F. A. Fontana, M. Zanoni, R. Roveda, Change prediction through coding rules violations, EASE'17, 2017, pp. 61–64.
[82] V. Lenarduzzi, V. Nikkola, N. Saarimki, D. Taibi, Does code quality affect pull request acceptance? an empirical study, Journal of Systems and Software 171 (2021) 110806.
[83] A. Trautsch, S. Herbold, J. Grabowski, A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects, Empir Software Eng 25 (2020) 51375192.
[84] S. Kim, M. D. Ernst, Which warnings should i fix first?, in: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 2007, p. 4554.
[85] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, S. Romano, On the accuracy of sonarqube technical debt remediation time, in: 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, pp. 317–324.
[86] M. T. Baldassarre, V. Lenarduzzi, S. Romano, N. Saarimki, On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube, Information and Software Technology 128 (2020) 106377.
[87] V. Lenarduzzi, V. Mandić, A. Katin, D. Taibi, How long do junior developers take to remove technical debt items?, in: 14th International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020.

[88] V. Lenarduzzi, F. Lomio, H. Huttunen, D. Taibi, Are sonarqube rules inducing bugs?, in: 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 501–511.

[89] J. R. Wright, K. Ali, L. N. Q. Do, Why do software developers use static analysis tools? a user-centered study of developer needs and motivations, IEEE Transactions on Software Engineering (TSE) (2020).

[90] X. Yang, J. Chen, R. Yedida, Z. Yu, T. Menzies, Learning to recognize actionable static code warnings (is intrinsically easy), 2021. arXiv:2006.00444.

[91] W. N. Oizumi, A. F. Garcia, L. da Silva Sousa, B. B. P. Cafeo, Y. Zhao, Code anomalies flock together: exploring code anomaly agglomerations for locating design problems, in: 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, 2016, pp. 440–451. URL: http://doi.acm.org/10.1145/2884781.2884868. doi:10.1145/2884781.2884868.

[92] W. Oizumi, A. Garcia, M. Ferreira, A. von Staa, T. Colanzi, When code-anomaly agglomerations represent architectural problems? an exploratory study, in: Brazilian Symposium on Software Engineering (SBES), 2014, pp. 91–100.

[93] S. Herold, An initial study on the association between architectural smells and degradation, in: European Conference on Software Architecture (ECSA), 2020, pp. 193–201.

[94] D. Sas, P. Avgeriou, U. Uyumaz, On the evolution and impact of architectural smellsan industrial case study, Empirical Software Engineering 27 (2022) 86.

[95] H. Mumtaz, P. Singh, K. Blincoe, Analyzing the relationship between community and design smells in open-source software projects: An empirical study, in: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2022, pp. 23–33.

[96] G. Samarthyam, G. Suryanarayana, T. Sharma, Refactoring for software architecture smells, in: 1st International Workshop on Software Refactoring, IWoR@ASE 2016, Singapore, Singapore, September 4, 2016, 2016, pp. 1–4. URL: https://doi.org/10.1145/2975945.2975946. doi:10.1145/2975945.2975946.

[97] S. Ganesh, T. Sharma, G. Suryanarayana, Towards a principle-based classification of structural design smells, Journal of Object Technology 12 (2013) 1:1–29. doi:10.5381/jot.2013.12.2.a1.

[98] S. Bergstrom, J. Boskovic, R. Mehra, Development of the adaptive reconfigurable control analysis, design, and evaluation (arcade) toolbox, 2003.