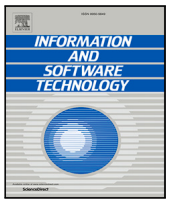
Information and Software Technology 199 (2026) 108283

Contents lists available at ScienceDirect

Information and Software Technology

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



Empirical study on the correlation among architectural smells

Matteo Bochicchio [a,*], Bartosz Walter [b], Francesca Arcelli Fontana [a]

[a] University of Milano-Bicocca, Italy
[b] Poznan University of Technology, Poland

ARTICLE INFO

Dataset link: https://zenodo.org/records/17828125

Keywords:
Architectural debt
Architectural smells
Software maintenance
Software evolvability
Empirical study

ABSTRACT

Context: Architectural smells are indicators of architectural technical debt and may compromise the maintainability and evolvability of software systems. Hence, it is important to carefully study and monitor them.
Objective: This study aims to investigate statistical co-locations, i.e., correlations among architectural smells affecting the same components, to assess whether specific patterns of smells frequently appear together, forming clusters.
Method: We constructed a large-scale dataset using an automated data collection pipeline with Arcan tool, analyzing 782 Java and Python projects. We applied Principal Component Analysis and Association Rule Mining to identify recurrent co-location patterns, finally illustrating some examples of patterns.
Results: The analyses reveal possible co-location patterns: dependency-related smells (Cyclic Dependency, Hub-like Dependency, Unstable Dependency) often cluster with hierarchy-related smells (Cyclic Hierarchy, Wide Hierarchy). Other smells, such as God Component and Deep Hierarchy, are infrequent and do not significantly contribute to dominant patterns. Association Rule Mining confirms these associations with high conviction but generally low support values.
Conclusion: Architectural smells do not occur only in isolation but also in recurring clusters that point to deeper structural issues. These findings suggest that architectural degradation may arise from interaction between smells rather than individual ones, highlighting the need for analysis of smell co-locations.

1. Introduction

Technical debt is a metaphor originally introduced by Cunningham to describe the cost resulting from suboptimal decisions at various levels, as an analogy for financial debt [1]. While it can be accumulated and serviced, e.g., by loans or deferred payments, it must eventually be repaid, and the cost of it must b not be greater than the expected profits. The technical debt metaphor captures suboptimal design- and implementation solutions that yield a benefit in the short term but may not meet current requirements, making future changes more complex and expensive [2]. While initially focused on the code, the term has since been broadened to encompass a wider range of activities, including software architecture, design, requirements, documentation, and testing. Technical debt may result from *conscious decisions*, e.g., to expedite the delivery of new features, or *inadvertently*, as a side effect of natural evolution, due to limited skills, knowledge, time constraints, or as a result of unforeseen design issues. Technical debt tends to grow when not properly addressed, leading to increased maintenance costs, reduced system adaptability, and a general decline in software quality [3]. Alves et al. [4] defined categories of technical debt based on their context. Among these, *architectural technical debt* is identified as the category with the greatest impact on the overall technical debt of a software project. It results from architectural decisions that violate design principles, negatively affecting performance, scalability, and maintainability [5]. Unlike code-level debt, architectural debt cannot be mitigated with minor edits. Conversely, it typically requires extensive refactoring that affects multiple system components. Therefore, *bad architectural choices* represent the leading source of technical debt [3]. Keeping this in mind, the concept of architectural technical debt is closely related to the concept of architectural smells. Architectural smells are symptoms of anomalies in a software system's architecture. The presence of such smells correlates with architectural erosion and a progressive decline in overall software quality. They also contribute to the increase of the project's technical debt [6].

In recent years, several studies tried to better understand this phenomenon by employing various research methodologies, with most of them adopting a quantitative data collection methodology based on static code analysis [7].

* Corresponding author.
E-mail addresses: matteo.bochicchio@unimib.it (M. Bochicchio), bartosz.walter@cs.put.poznan.pl (B. Walter), francesca.arcelli@unimib.it (F.A. Fontana).

https://doi.org/10.1016/j.infsof.2026.108283
Received 18 December 2025; Received in revised form 8 July 2026; Accepted 28 July 2026
Available online 4 August 2026
0950-5849/© 2026 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

This paper presents an empirical study to investigate whether meaningful *co-locations* exist among specific types of architectural smells. To our knowledge, a substantial body of work has investigated only individual effects of architectural smells on maintainability. In the literature, recent studies have examined the correlations between architectural smells and other software quality and maintainability characteristics, such as design smells [8], design patterns [9], code smells [10], and test smells [11]. However, this approach may limit the ability to understand or predict maintenance problems when the interactions among smells are not taken into account. Therefore, investigating correlations among architectural smells represents a novel contribution. The same observation was made by Yamashita et al. concerning code smells [12], and other studies supported it [13].

The remainder of this paper is organized as follows: Section 2 introduces the Arcan tool used to construct our dataset and describes the architectural smells considered (detected by the tool). Section 3 outlines the data collection pipeline. Section 4 presents the study design, while Section 5 reports the results. Then, Section 6 discusses the findings, and Section 7 details the study's limitations and threats to validity. Section 8 summarizes the most relevant related work in the literature and compares them with those analyzed in our study. Finally, Section 9 presents the conclusions and outlines possible directions for future work.

All data related to the empirical study, dataset, tests, scripts, and the obtained results are publicly available in the replication package [14].

2. Background on arcan

Arcan 2.11.1 is the newer release of the publicly available[1] tool for automatic technical debt detection through static source code analysis, initially published by Arcelli et al. [15]. Out of various types of technical debt [3,4], Arcan 2.11.1 focuses specifically on architectural technical debt and related architectural smells (AS) [16,17]. As depicted in Table 1, it detects seven AS:

- **Hub-like Dependency (HL):** (detected on classes and packages) refers to a component with many (outgoing and incoming) dependencies with a large number of other components [5,18].
- **Cyclic Dependency (CD):** (detected on classes and packages) refers to a component that is involved in a chain of relations that breaks the desired acyclic dependency structure of the system [18].
- **Unstable Dependency (UD):** (detected on packages) describes a component that depends on other subsystems that are less stable than itself, according to the Instability metric [18,19].
- **God Component (GC):** (detected on packages) This smell represents a component that is considerably larger in size (i.e., LOC) than other components in the system [18].
- **Cyclic Hierarchy (CH):** (detected on classes) This smell occurs when a super-type depends on one of its subtypes, creating a bidirectional relationship that breaks the intended hierarchy [5].
- **Deep Hierarchy (DH):** (detected on classes) This smell occurs when an inheritance tree becomes excessively long, making the system harder to understand, maintain, and test [5].
- **Wide Hierarchy (WH):** (detected on classes) This smell occurs when an inheritance hierarchy is too broad, suggesting the absence of intermediate abstraction layers [5].

These smells have a significant impact on software quality and architectural technical debt. As highlighted by Le et al. [20] and Mo et al. [21], both the God Component and the presence of three dependency-related AS (CD, UD, HL) strongly influence maintainability and architectural debt. Furthermore, as emphasized by Suryanarayana et al. [5], the three hierarchy-related smells (CH, DH, WH) severely affect changeability and testability, making the code more complex and leading to unpredictable behavior.

Arcan is capable of detecting AS at two levels of granularity, labeling components as either "containers" or "units". The term container refers to components at the highest level of abstraction (package-level), while unit refers to components at the lowest level of abstraction (class-level). In the context of the programming languages considered in this study, a container corresponds to a Java package or, in Python, to either a Python package or a directory. Conversely, a unit corresponds to Java class or, in Python, to either a Python module or a file.

Table 1 reports the AS detected, the metrics that the tool can compute regarding AS, and architectural technical debt. For each AS type, Arcan 2.11.1 detects instances of the smell and assigns each instance a severity value, representing the criticality of the smell, and estimates the overall amount of technical debt. Arcan 2.11.1 is also capable of computing a set of metrics to estimate Technical Debt (TD) as an index, obtained by summing the Architectural Technical Debt Index (ATDI) of each detected smell.

To support the validity of the AS detected by Arcan, two previous studies assessed the Arcan accuracy for dependency smells. Arcelli et al. [22] performed manual validation on two industrial projects involving three professional software designers, reporting 100% precision for CD, HL, and UD; observed false negatives were attributed to components outside the tool's analysis scope. A further industrial evaluation [23] assessed 19 smell instances (including CD, HL, UD, and GC) through a developer survey, finding that all CD and UD instances were classified as true positives. Moreover, GC exhibited two false positives out of 4 instances and the single HL instance at the package level was identified as a false positive, as the affected package contained utility classes intentionally designed to be accessed system-wide. These false negatives, however, have been manually verified as conscious decisions made by developers, who choose to introduce them for various reasons, and can therefore be recognized as self-admitted technical debt. Regarding the hierarchy smells, Arcan's implementation uses the same detection rules as the Designite tool, developed by Sharma et al. [24], used as well in many previous studies [8,25–27], and also used to construct the QScored dataset [28]. These rules follow the hierarchy smells definitions given by Suryanarayana et al. [5].

3. The data collection pipeline

To build our dataset, we leveraged and extended a data collection pipeline that we introduced in an our previous work [29]. The pipeline was specifically designed to support researchers in constructing large datasets for empirical studies. Our pipeline automatically builds the dataset of AS using Arcan once the projects to be monitored are defined, and is capable of updating the dataset on a regular basis. We extended the pipeline to make it compatible with the latest version of the tool. The most significant difference is that the new dataset also includes information on hierarchy-related smells, which were not supported in the previous version of Arcan. For completeness, we also provide a brief overview of the pipeline's architecture and functionality.

In summary, the data collection pipeline automatically monitors a set of GitHub repositories, retrieves new releases or commits, and schedules analyses with Arcan. Implemented with Apache Airflow, it orchestrates three main workflows: (i) *Ingestion*, which selects and updates project versions to analyze; (ii) *Execution*, which runs Arcan in parallel Docker containers to parse source code, detect AS, and compute metrics; and (iii) *Generation*, which consolidates the results into a database. The pipeline is fully automated, scalable, and designed to facilitate the construction of large, reproducible datasets for empirical research. For further implementation details, the pipeline's source code is publicly available.[2] Our pipeline can also be configured to operate with other static analysis tools that support exporting their results.

[1] https://github.com/Arcan-Tech/arcan-2-cli-trial

[2] See https://github.com/Arcan-Tech/arcan-pipeline.git.

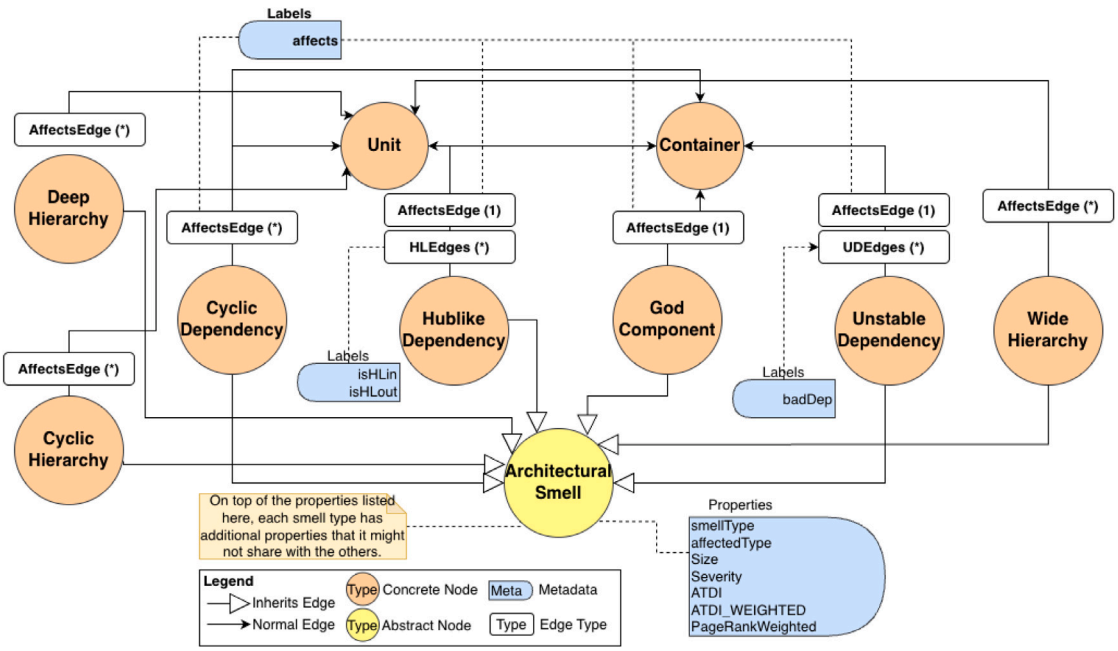


Fig. 1. Smells dependency graph schema.

The dataset produced by the pipeline is structured as a dependency graph generated by Arcan. In the graph, nodes represent source code artifacts, such as packages, classes, and functions, and edges capture their relationships, including membership and dependencies. Four main node types are defined: units, containers, functions, and smells. Containers, units, and functions are collectively referred to as components and share common attributes (e.g., qualified name, file path, and software metrics). The final dataset is a flattened representation of these graphs, where each node or edge corresponds to a row in the database and each property to a column.

Fig. 1 shows the information related to smell-type, along with the relevant properties associated with smell nodes. The figure provides a clear overview of the dataset structure produced by the tool, and allows to understand the starting point from which the dataset was manipulated in order to prepare it for automated analyses, also with a view to reproducibility. The figure also depicts the edge representing the 'affects' relationship, which is of primary interest for the purposes of this study. Depending on the granularity of the smell, the 'affects' edge connects the smell node to either a unit-type node (i.e., a class) or a container-type node (i.e., a package). These edges indicate that an instance of a given smell type affects a specific component. In the case of instances affecting multiple components (e.g., Cyclic Dependency), multiple edges will be present.

The dataset is storeg in an DuckDB database [30] that represents the nodes and edges of the dependency graph produced by Arcan as two tables. The nodes table's records are either components or smells in a specific version, with each field representing the software metrics and smell characteristics, as reported in Table 1 and Fig. 1. Similarly, each record in the edges table represents a single edge in the graph with its properties as fields, including which nodes they connect and their weight. The complete database is available within the replication package [14].

Table 1
The TD principal metrics, list of AS and a set of metrics computed by Arcan 2.11.1 that we considered in our work.

Name	Granularity	Value type	Ref.
Technical debt principal			
Arch. tech. debt index (ATDI)	Smell, Project	$\mathbb{N}$	[32]
Arch. tech. debt hours	Smell, Project	$\mathbb{N}$	[32]
Arch. tech. debt score	Project	$[0, 1]$	[29]
Arch. tech. debt rating	Project	$\{A, B, C, D, F\}$	[29]
Architectural smells			
Hub-like Dependency (HL)	Package, Class	–	[22]
Cyclic Dependency (CD)	Package, Class	–	[22]
Unstable Dependency (UD)	Package	–	[22]
God Component (GC)	Package	–	[18]
Cyclic Hierarchy (CH)	Class	–	[5]
Deep Hierarchy (DH)	Class	–	[5]
Wide Hierarchy (WH)	Class	–	[5]
Smell characteristics[a]			
Affected component type	All smells	[b]	[33]
Severity	All smells	$(0, 10)$	[32]
Software metrics			
Fan In	Package, Class	$\mathbb{N}$	[34]
Fan Out	Package, Class	$\mathbb{N}$	[34]
Lines of code	Package, Class	$\mathbb{N}$	–

Note: we only use Java-specific terminology (e.g., package) for brevity's sake, the metric applies to the respective counterparts of other programming languages. All the details of the metrics are available in the referenced literature cited in the table.

[a] Also known as smell properties.

[b] $\{Package, Class\}$.

4. Study design

4.1. Goal and research questions

This study aims to investigate possible correlations among specific AS, or whether they are independent.

Following the Goal-Question-Metric approach [31], the goal can be formulated as follows:

Study the statistical correlation between architectural smells in order to investigate whether the presence of certain smells is systematically associated with the presence of others.

The goal can be summarized in the following two research questions:

***Research Question 1 (RQ1)**: Are there specific architectural smells that tend to be co-located, i.e., to affect the same component?*

***Research Question 2 (RQ2)**: Are there systematic co-location patterns among architectural smells that tend to manifest across diverse software projects?*

From now on, by using the concept of a *component*, we refer to either *containers* (package-level) or *units* (class-level), depending on the type of smell being analyzed.

Answering **RQ1** allows us to determine whether certain AS are more likely to affect the same component and to determine whether specific smells consistently co-locate at the component level, thereby revealing systematic interactions among design flaws. **RQ2** extends this perspective by examining whether such co-location patterns are systematic across different projects, thus distinguishing project-specific anomalies from generalizable phenomena. Together, these research questions aim to advance the understanding of how AS interact at different granularity levels and provide insights that can support more effective architectural debt management.

4.2. Analyzed projects

To conduct this study, we employed the data collection pipeline described in Section 3. To retrieve the projects to be analyzed, we first retrieved the URLs and metadata of the repositories using the GitHub GraphQL API.[3] This process was automated through a Python script. To overcome the API's limitation of 1000 items per search operation, the script implements a pagination strategy based on predefined star count intervals (buckets). The following intervals were used: [100000, 50000], [49999, 20000], [19999, 10000], [9999, 5000], [4999, 2000], [1999, 1000], [999, 500], [499, 100].

The repository search process was guided by the following filters:

- Only projects written in Java and Python were considered in this study. These two languages were selected not only because they are both supported by Arcan, but also to allow the analysis to capture potential differences in AS across programming paradigms. In particular, Java is a fully object-oriented language, whereas many Python projects tend to adopt a procedural or hybrid programming style [35]. Including both languages helps mitigate threats to validity by ensuring that the findings are not tied to a single programming paradigm.
- Projects with a value greater than or equal to 100 stargazers. The number of stargazers provides an indicative measure of the repository's relevance and contribution within the GitHub community [36]. It helps ensure the selection of open-source projects that are widely used and community-supported, thereby reducing potential noise in the dataset. This threshold has already been employed in the literature by Han et al. [37].

After this initial project search phase, we identified a total population of **3,089** projects.

Subsequently, to select *high-quality* projects and discard those that could introduce noise, we analyzed all the retrieved repositories using the tool RepoQuester.

RepoQuester tool, presented by Boyalakuntla et al. [38], is a re-engineered and extended version of the previously proposed RepoReaper tool [39]. The tool supports researchers in automatically identifying GitHub repositories that contain engineered software projects, that are projects that exhibit concrete evidence of software engineering typical practices (e.g., have good issue handling, testing, license, documentation, and community support). RepoQuester analyzes and evaluates each repository along nine key dimensions that reflect typical software engineering practices and assigns to each dimension a corresponding score: Continuous Integration, Documentation, History, License, Unit Testing, Community, Issues, Releases, Pull Requests Ratio.

Repositories that score low across these dimensions generally exhibit *lower quality* compared to those with higher scores.

[3] See https://docs.github.com/en/graphql/overview/about-the-graphql-api.

Table 2
Distribution of the metrics for the projects included in the study grouped by programming language.

Metrics	Java	Python
Number of projects	356	426
LOC median	22881.5	19407.0
Community median	4	6
Documentation median	0.181	0.147
History median	15.841	24.351
Unit Test median	0.008	0.126
Pull Request Ratio median	0.540	0.500
Number of 1 in License	249 (69.9%)	229 (50.2%)
Number of 1 in Releases	249 (69.9%)	262 (61.5%)

The Issues dimension was excluded from the analysis because the RepoQuester tool consistently returns zero for all analyzed projects. Similarly, the variable related to Continuous Integration was disregarded, as its distribution in the initial dataset was heavily skewed toward zero.

Considering the remaining seven dimensions, we applied the following filter:

- only repositories that scored more than 0 in at least *five* out of the seven dimensions were retained. This criterion is inspired by the one used by Sharma et al. [8]. However, we opted to relax the original threshold to obtain a more balanced and uniform dataset, ensuring that the distribution of all dimensions remains comparable, and to avoid bias.
- we retained only those projects with at least 10,000 LOC and more than 10 files, to focus on sufficiently large projects and to ensure a sufficient number of AS. By files, we refer to `.java` or `.py` files (source code files), depending on the programming language. Similar threshold values have been used in previous studies [29,40,41].

After applying this filter, **782** projects remained (25,3% of the initial population).

Table 2 provides an overview of the project population included in the study, grouped by programming language. It reports the number of projects for each language, the median value of LOC, and the computed dimensions.

4.3. Collected architectural smells

Table 3 reports the number of unique instances of AS of each type identified in the dataset, grouped by programming language. The dataset contains a total of 156,920 unique instances of AS. For this study, only the latest version available at the time of the analysis was examined for each project through the pipeline, as there was no need to track multiple versions. Cyclic Dependencies are the most prominent type of AS in terms of the number of instances. On the other hand, God Components and Deep Hierarchy are the least frequently observed smells. This is mainly because God Components are size-based smells, and large components are relatively uncommon. Similarly, Deep Hierarchy is triggered by excessively deep inheritance chains, which are rare, as it is uncommon for classes to inherit a large number of methods due to the presence of multiple supertypes. It is also interesting to note that no Python project has an Unstable Dependency smell. The result stems from the fact that many Python projects are written in a procedural style rather than an object-oriented one. As such, the metrics used to calculate Unstable Dependency smells (defined by R. C. Martin [19]) are less meaningful in these cases. The same applies to Cyclic Dependency, but only at class level.

UD instances and CD instances at the class level in Python projects are absent due to Python's programming paradigm. Moreover, it should be noted that programming styles differ across languages and vary

Table 3
Total number of architectural smells identified across the project population.

Architectural smell	Java	Python	TOTAL
Cyclic Dependency	77,672	21,314	**98,986**
Hub-like Dependency	1819	1416	**3235**
God Component	190	197	**387**
Unstable Dependency	1144	0	**1144**
Cyclic Hierarchy	22,712	22,750	**45,462**
Wide Hierarchy	5082	2464	**7546**
Deep Hierarchy	82	78	**160**
TOTAL	**108,701**	**48,219**	**156,920**

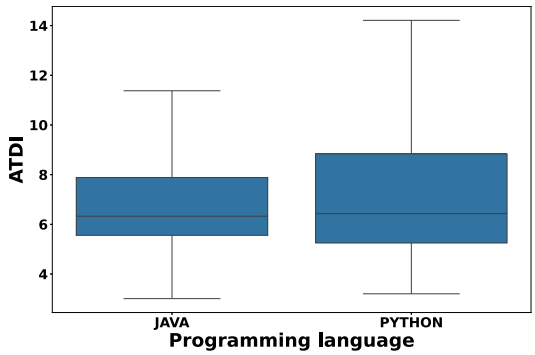


Fig. 2. Distribution of ATDI by programming language.

considerably depending on the developer's background, with Python code written by programmers with prior Java experience differing substantially from that written by native Python developers [42]. Dependencies in Python also have a different nature compared to Java: Python relies on dynamic imports and runtime dependency resolution, which makes certain dependency-related smells less amenable to static detection and, in some cases, less meaningful to measure.

It is worth noting that hierarchy-related smells (CH, WH, DH) were detected in substantial numbers in Python projects, showing that the Python projects included in our dataset do make meaningful use of object-oriented mechanisms.

The dataset is generally imbalanced, as the number of instances varies significantly across different AS. This imbalance may pose a threat to validity.

To assess representativeness, we compared our smell distribution with prior Arcan-based empirical studies on AS in open-source projects and observed similar trends, particularly the dominance of Cyclic Dependency [8–10]. Overall, despite the imbalance, the dataset can be considered sufficiently representative.

Fig. 2 illustrates the distribution of the ATDI values across programming languages. The dataset includes AS with a wide range of ATDI values, allowing the analysis to account for smells of varying levels of complexity. Further information on the project population can be found in the replication package [14].

4.4. Collected variables

For our study, we introduce the following definition:

Co-located architectural smells are defined as two or more AS that are detected within the same component at the same time. Therefore, to address the RQs, we aim to investigate the correlation between co-located AS.

To achieve that, it is necessary to collect, for each AS type, the number of instances affecting each class and package. In summary, for *every component* in a project, we collect the following variables:

- **Count**: This variable is used to track the number of occurrences of each type of AS affecting the component. There are seven variables in total, one for each smell type. For example, the variable *HL count* may take the value 0 if no HL smell affects the component, or the value three if exactly three instances of a HL smell affect the component.
- **Presence**: This binary variable is used to indicate whether at least one instance of a given AS type affects the component. There are seven variables in total, one for each smell type.

These variables will be computed depending on whether the component is a class or a package. Specifically, HL count, CD count, UD count and GC count variables were considered at the package level, while HL count, CD count, CH count, DH count and WH count variables were considered at the class level. The same reasoning applies to the presence variables.

Using these variables, for *every project* we construct four matrices:

- **Class Co-location Matrix (CCM)**: consisting of a maximum of 5 columns, one for each *count* variable corresponding to class-level AS, and with the number of rows equal to the number of classes in the project. It is not guaranteed that all five class-level AS are present in each project. The matrix is constructed so that the number of columns corresponds only to the AS instances that affect the projects' classes. As a result, no column will contain only zeros.
- **Package Co-location Matrix (PCM)**: consisting of a maximum of 4 columns, one for each *count* variable corresponding to package-level AS, and the number of rows equal to the number of packages in the project. As with the CCM matrix, there are no columns containing only zeros, and the number of columns varies from project to project depending on the present AS.
- **Binary Class Co-location Matrix (BCCM)**: The structure is the same as the CCM matrix, but it considers the *presence* variables instead of the count variable, thus resulting in a binary matrix.
- **Binary Package Co-location Matrix (BPCM)**: The structure is the same as the PCM matrix, but it considers the *presence* variables instead of the count variable, thus resulting in a binary matrix.

For this study, the following confounding variables have been identified and must be mitigated:

- **unit/module size**: This is often a confounding variable when analyzing AS, as larger components are more likely to be affected by smells, potentially by multiple types simultaneously [29,43]. The size of the component is measured in terms of its LOC. Arcan computes the LOC metric at both the package and class level, excluding blank lines and commented lines from the count.
- **Fan-In**: This variable represents the number of incoming dependencies to a given component [34]. AS relates to dependency issues are more likely to occur in components with high Fan-In values. For this reason, mitigating this confounding variable also helps address the imbalance in the distribution of smell instances, as the most frequent smells, such as Cyclic Dependency, are strongly dependency-based [6,21,43]. This metric is computed by Arcan at both the class and package level.
- **Fan-Out**: This variable represents the number of outgoing dependencies from a class [34]. Similar reasoning as for Fan-In applies: components with high Fan-Out values are more likely to exhibit AS. This metric is also computed by Arcan at both the class and package level.

These variables were collected in the dataset and associated with the components in the CCM and PCM matrices. The BCCM and BPCM matrices do not need to incorporate information about confounding variables, since the mere presence of a smell instance, expressed as a binary value, does not require normalization. However, due to the way Arcan computes the LOC for Python classes, we decided not to consider the relative LOC value in our analysis. This decision stems

from a limitation of the tool, which is not always able to correctly compute the LOC for Python classes. This issue may have been caused by a fundamental difference between the two programming languages: in Java, each `.java` file defines exactly one public class with the same name as the file, whereas in Python, a single `.py` file can contain multiple class definitions with arbitrary names unrelated to the file name. As a result, Arcan often miscalculates the LOC value for Python classes, frequently assigning a value of zero, which in turn makes it impossible to rely on the LOC value. This issue, however, does not occur with Python packages.

Moreover, as indicated in Table 3 and explained in Section 4.3, no Python project in the dataset is affected by UD smells. Consequently, the PCM and BPCM matrices for Python projects do not include the column corresponding to the UD count and the UD presence variables. For the same reason, Arcan did not detect any CD at the class level. As a result, the CCM and BCCM matrices for Python projects do not include the CD count and CD presence variable.

4.5. Statistical analysis

This section presents the analyses conducted and the statistical methods we employed.

4.5.1. Principal component analysis

To answer **RQ1** and investigate the correlation of AS, we chose to apply *Principal Component Analysis (PCA)*, using orthogonal rotation (varimax). PCA is a statistical technique used to reduce the dimensionality of a dataset while preserving as much of the original variance as possible. It transforms the original correlated variables into a new set of uncorrelated variables, known as principal components, which are ordered such that the first few retain most of the variation from the original dataset. PCA is commonly employed for exploratory data analysis, noise reduction, and feature extraction, and it facilitates the visualization and interpretation of high-dimensional data [44]. In the context of our study, PCA is employed to uncover latent structures among AS by identifying patterns of co-location across classes or packages. This allows us to group smells that tend to appear together within the same component. The rationale for using PCA lies in its successful application in previous empirical studies investigating correlations among code smells [12,45], where it proved effective in identifying recurring patterns. Moreover, PCA is a well-established and widely recognized statistical technique. We adopted the varimax rotation in the PCA: as noted by Field [44], this enhances the clarity of the extracted components, making their interpretation more straightforward.

For our PCA-based analysis, for *each matrix*, we performed the following steps:

1. First, we discarded the matrices for which the application of PCA would not be meaningful. Specifically, a matrix was excluded if it contained information on fewer than two AS, had fewer than four rows, or if the number of rows exceeded the number of columns. This choice is consistent with established PCA guidelines proposed by Jolliffe [46], which require a non-singular correlation matrix and a sufficient number of observations to obtain stable and interpretable components. In addition, all singular matrices were excluded, since the implementation employed requires invertibility as a necessary condition.[4]
2. Subsequently, we standardized the residual values using z-score normalization to ensure that all smell dimensions had zero mean value and unit variance. This step is essential because PCA is sensitive to the scale of input features; without standardization, variables with higher variance (regardless of interpretive importance) would disproportionately affect the resulting principal components.
3. Then, an initial PCA (with the varimax rotation) was performed to compute the eigenvalues of the correlation matrix and determine the number of significant components according to the Kaiser criterion: all components with eigenvalues greater than 1 were retained [47].
4. Finally, PCA was re-applied to the standardized data, with the number of components set according to the Kaiser criterion. This final analysis produced the factor *loadings matrix*, which reflects the strength and direction of the association between each AS and each extracted principal component. The loadings can be interpreted as the correlation coefficients between the original variables (i.e., the occurrences of a smell) and the principal components, and are used to identify which smells contribute most to each latent dimension uncovered by the PCA. High absolute values in the loading matrix indicate a strong influence of a smell on the corresponding component, thereby facilitating the interpretation of underlying patterns of co-location. AS were considered strongly associated with a component if the absolute value of their loading was $\geq .5$. According to Hair et al. [48], loadings $\geq .5$ are considered practically significant.

Before performing the PCA analysis, we first applied (i) a regression approach to mitigate the confounding variables identified in Section 4.4, and (ii) adequacy tests to justify the application of PCA.

4.5.2. Regression-based metric adjustment

To mitigate the impact of confounding variables identified in Section 4.4, we applied a multiple linear regression approach based on *Ordinary Least Squares Regression (OLS)*.

The OLS regression is a fundamental statistical method used to model the linear relationship between a dependent variable and one or more independent variables. The technique estimates the model parameters by minimizing the sum of squared residuals, that is, the differences between the observed values and those predicted by the model. This minimization ensures that the resulting line (or hyperplane, in the case of multiple predictors) provides the best possible fit in terms of least squared error. Due to its simplicity, interpretability, and computational efficiency, OLS remains one of the most widely adopted techniques in both exploratory data analysis and predictive modeling [49].

Specifically, for each AS, we regressed its values against three key variables: LOC, Fan-in, and Fan-out. The residuals obtained from these regressions represent the portion of each count variable that is not explained by the size or dependency profile of the component. In other words, they capture the deviation of each count variable from what would be expected based solely on the structural attributes of the component. By isolating this residual component, we can focus the PCA on the intrinsic relationships among smells, rather than on variance driven by component size or dependency factors.

These residual values were then used as input for the subsequent PCA.

Therefore, the OLS regression was applied to both the CCM and PCM matrices for each project.

4.5.3. Adequacy tests

Before applying PCA to the matrices resulting from the OLS regression, we conducted adequacy tests to assess whether the variables were sufficiently correlated to justify the use of factor analysis. For this purpose, we employed the *Kaiser–Meyer–Olkin (KMO)* test and *Bartlett's test of sphericity*, as suggested by Field [44].

The KMO test evaluates the adequacy of the sample for conducting factor analysis or PCA. Specifically, it quantifies the proportion of variance among the variables that may be attributed to the underlying

[4] We used the FactorAnalyzer python library to perform the adequacy tests and conduct the PCA. See the documentation at https://factor-analyzer.readthedocs.io/en/latest/introduction.html for more details.

common factors. The KMO statistic ranges from 0 to 1, with higher values suggesting that the correlation patterns are sufficiently compact to justify the application of dimensionality reduction techniques. A KMO value greater than 0.5 is typically regarded as acceptable, while values close to 1 indicate a high degree of sampling adequacy for PCA [50].

Bartlett's test is a statistical procedure used to assess whether some independent samples originate from populations with equal variances [51]. The hypotheses associated with the tests are as follows:

- Null hypothesis (H_0): The variances in the populations are equal.
- Alternative hypothesis (H_1): At least one population has a different variance.

The test is conducted at a significance level $\alpha = 0.001$.

For these tests, the input data consisted of the CCM and PCM matrices, which contained the residuals from the OLS regression, and the statistical tests were performed separately for each project.

We applied the following criterion for considering a CCM or PCM matrix suitable for PCA:

$$kmo > .5 \vee pval < .001 \quad (1)$$

where kmo refers to the result of the KMO test and $pval$ indicates the p-value associated with Bartlett's test of sphericity. Matrices not satisfying this condition are considered unsuitable for PCA and therefore excluded from further analysis. These thresholds on adequacy tests have already been adopted in other empirical studies [12,45].

4.5.4. Association rules mining

In order to answer **RQ2**, we adopted *Association Rule Mining (ARM)* as a means to uncover systematic patterns of co-location among AS. The input to the ARM analysis consists of the BCCM and BPCM matrices. ARM is an unsupervised learning technique aimed at identifying local patterns that capture attribute-value conditions frequently co-occurring within a dataset [52]. Formally, let $I = \{i_1, \ldots, i_n\}$ denote a set of n binary attributes, referred to as *items*, each representing the presence of a specific property in the element under analysis. Let $T = \{t_1, \ldots, t_m\}$ be a set of m *transactions*, where each transaction corresponds to one element of the dataset and is defined as a subset of I. An *association rule* is then defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. In our work, the set T comprises all the components present in a specific project, while each item in the set I indicates the presence of a particular smell in that project. In this work, the association rules were generated using the Apriori algorithm [53]. We chose to employ this algorithm because it is efficient and widely used in empirical studies within the field of software engineering [54–56]. In this study, we consider three metrics to evaluate the extracted association rules: *support*, *confidence*, and *conviction*. Support quantifies the importance of a rule with respect to the entire dataset, indicating how frequently the antecedent and consequent occur together. Confidence reflects the accuracy of a rule, measuring the proportion of cases in which the consequent is observed given that the antecedent holds. Both metrics take values in the $(0, 1)$ range, with higher values suggesting stricter rules. However, support and confidence alone may not always provide a complete assessment, particularly when the dataset is relatively small or consists primarily of binary attributes. To complement these measures, we also employ conviction, which combines support and confidence into a single metric that evaluates how often a rule would be incorrect if the association were due to chance. Conviction ranges from 0.5 to ∞, where a value of 1 indicates independence between antecedent and consequent, values below 1 suggest negative dependency, and values above 1 indicate positive dependency [54]. For our study, we also consider the number of projects in which each rule is extracted, as this allows us to highlight the most recurrent associations across projects.

Before generating the association rules, BCCM and BPCM matrices containing fewer than two smells per level of abstraction were excluded from the analysis, as they were not meaningful.

Table 4
Overall distribution of the adequacy statistical test results.

	KMO	Bartlett chi square	Bartlett p value
Java class-level AS			
mean	0.431	82.990	$5.127 \cdot 10^{-2}$
std	0.122	170.615	$1.673 \cdot 10^{-1}$
median	0.460	36.555	$2.804 \cdot 10^{-6}$
Java package-level AS			
mean	0.495	26.869	$1.524 \cdot 10^{-1}$
std	0.079	87.469	$2.401 \cdot 10^{-1}$
median	0.500	10.537	$2.303 \cdot 10^{-2}$
Python class-level AS			
mean	0.491	37.732	$1.102 \cdot 10^{-1}$
std	0.085	113.290	$2.061 \cdot 10^{-1}$
median	0.500	14.857	$1.852 \cdot 10^{-3}$
Python package-level AS			
mean	0.492	11.862	$2.594 \cdot 10^{-1}$
std	0.051	26.201	$3.210 \cdot 10^{-1}$
median	0.500	3.910	$6.886 \cdot 10^{-2}$

5. Results

This section presents the results of the analyses conducted in the empirical study, as described in the previous section.

5.1. Answering RQ1

After filtering the matrices, PCA was successfully conducted on matrices from **300** Java projects (84% of the initial population of Java projects) and **268** Python projects (63% of the initial population of Python projects). For each project, either the CCM matrix, the PCM matrix, or both were analyzed.

Table 4 presents the overall results of the adequacy tests, broken down by programming language and by smell levels (class-level or package-level). As shown in the table, Java class-level matrices yielded the most statistically significant results in the Bartlett's test, as indicated by their notably low median p-value. Overall, matrices from Python projects were discarded at a higher rate than those from Java projects, likely due to differences in programming paradigms between the two languages. This is reflected in the lower number of retained Python projects and the generally less significant adequacy test results. Another noteworthy observation is that, on average, CCM matrices associated with class-level smells were more frequently deemed adequate than PCM matrices associated with package-level smells.

Figs. 3 and 4 show the PCA results for Java and Python projects, respectively, grouped by level (class or package). The intensity in the heatmap represents the number of projects in which a given smell has a strong loading on a specific principal component, i.e., its absolute loading value is greater than or equal to 0.5 [50]. It is important to emphasize that the components represented in the two heatmaps are those selected according to the Kaiser criterion presented in Section 4.5.1. The analysis is based on the residuals of each smell after controlling for confounding variables such as size, Fan-in, and Fan-out through regression.

Concerning Java projects, the heatmap in Fig. 3(a) illustrates the frequency with which each AS exhibits a strong loading on the first three principal components (C1, C2, and C3, those that meet the Kaiser criterion) derived from the PCA, aggregated across all class-level matrices. The results clearly indicate that the first component (C1) captures a dominant structural pattern, marked by the frequent co-location of CD, WH, and CH. Specifically, CD shows a strong loading on C1 in 165 projects (55%), WH in 164 (55.67%), and CH in 98 (32.67%); highlighting a recurrent architectural cluster that links dependency cycles with hierarchical anomalies at the class level. There is also a minor contribution from HL smell. This co-loading is particularly meaningful

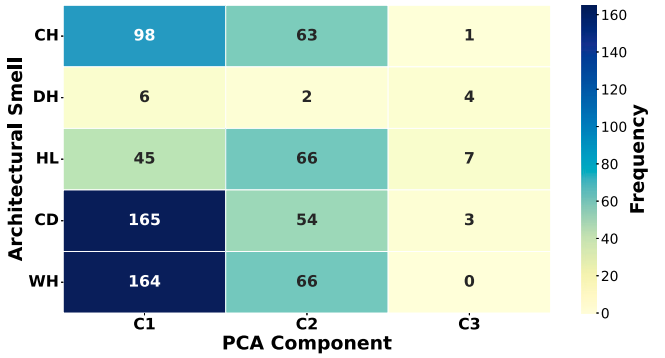
given that CD is the most frequent smell in the dataset; the fact that WH and CH load on the same component despite being less frequent suggests that these smells are not only individually prevalent but also structurally interrelated. Such a pattern might indicate that complex or poorly layered class structures are more prone to developing both cyclic dependencies and irregular hierarchies. The second component (C2) reflects a similar or complementary pattern. CD, WH, CH still load strongly in a notable number of projects (54, 66, and 63 respectively, which means 18%, 22% and 21% of Java projects), while HL also shows a substantial contribution, with strong loadings in 66 projects (22%). The third component (C3), on the other hand, does not display strong associations with any particular smell. The number of projects in which a smell loads on C3 remains consistently low (e.g., HL loads in only 7 projects), suggesting that this component likely captures residual variance or project-specific noise rather than meaningful latent structure.

At the package level, the heatmap in Fig. 3(b) summarizes the frequency of strong loads on the first two principal components (C1 and C2). The principal component C1 is clearly dominated by CD, HL, and UD, which load strongly in 135 (45%), 95 (31,67%), and 70 (23,33%) projects, respectively. This may indicate a consistent structural pattern where these smells co-locate in packages with highly entangled dependency structures. In contrast, the second component (C2) does not exhibit strong or consistent loadings for any particular smell, suggesting it may represent weaker or more project-specific patterns. GC shows minimal presence overall (28 projects on C1, 10 on C2), confirming that it does not systematically co-locate with dependency-related smells and likely reflects isolated design issues rather than broader structural patterns.

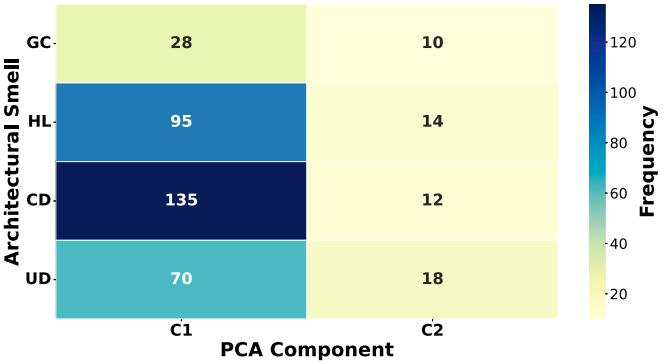
Turning over to the Python projects, the heatmap in Fig. 4(a) illustrates the number of Python projects in which each class-level AS loads strongly on the C1 and C2 principal components derived from the PCA. The results show that the C1 captures a dominant latent pattern characterized by the strong co-loading of three smells: CH, WH, and HL. Specifically, CH loads strongly on C1 in 189 (70.52%) projects, WH in 130 (48.51%) projects, and HL in 128 (47.76%) projects. This co-location may suggest that C1 reflects a structural syndrome of architectural degradation where components are affected by complex, deeply nested hierarchies (CH and WH) and centralized dependency structures (HL). The second component (C2) does not exhibit any dominant smell loadings. A few occurrences are observed (e.g., WH in 19 projects, HL in 14), but these appear to reflect project-specific variance or weak residual structures not representative for broader architectural patterns.

Finally, as shown in Fig. 4(b), the PCA at the package level for Python projects reveals a single dominant co-location pattern of AS. The C1 principal component is primarily associated with CD and HL, which load strongly in 71 (26.49%) and 56 (20.89%) projects, respectively, while GC also appears in 33 (12.31%). This may suggest that dependency entanglement and centralization, often coupled with overloaded packages, are frequent architectural issues in Python systems. The absence of consistently high smell loads on the C2 component suggests that a single latent factor captures most of the structural variance.

Overall, the PCA analyses across Java and Python systems, at both class and package levels, revealed the presence of latent co-location patterns among AS. At the class level, a recurring pattern that emerges links dependency-related issues, such as CD in Java and HL in Python, with design flaws in inheritance-related class hierarchies, including CH and WH, which are indicators of poorly structured or incomplete inheritance schemes. These combinations collectively point to potential threats to system changeability, extensibility, and reliability. At a higher level of abstraction, HL and CD appear to constitute another latent pattern. Components with a high number of dependencies may also be affected to cyclic dependencies, indicating an underlying structural fragility in the dependency architecture. In contrast, some AS, like GC and DH, are present less frequently, and this could indicate that they



(a) Class-level AS



(b) Package-level AS

Fig. 3. Heatmaps showing the frequency of strong associations between AS and PCA components across Java projects ($|loading| \geq .5$).

are not a key factor in the dominant co-location structures of smells and a distinct structural nature, at least within the analyzed projects.

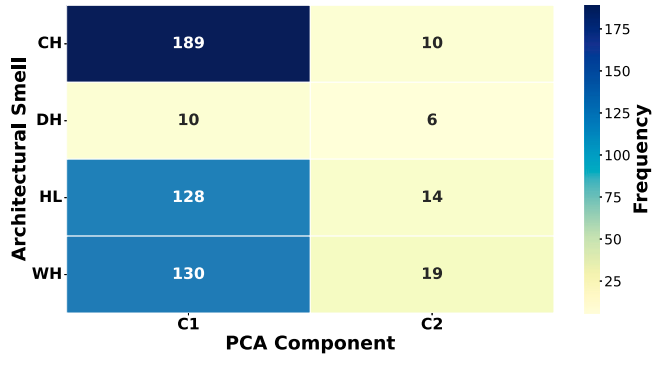
RQ1: Are there specific architectural smells that tend to be co-located, i.e., to affect the same component?

Answer: The PCA revealed that certain AS tend to co-locate across many projects. Dependency-related smells (CD, HL, UD) frequently cluster with hierarchy-related ones (CH, WH), forming latent patterns of structural fragility. In contrast, smells such as GC and DH appear rarely and do not shape dominant co-location structures.

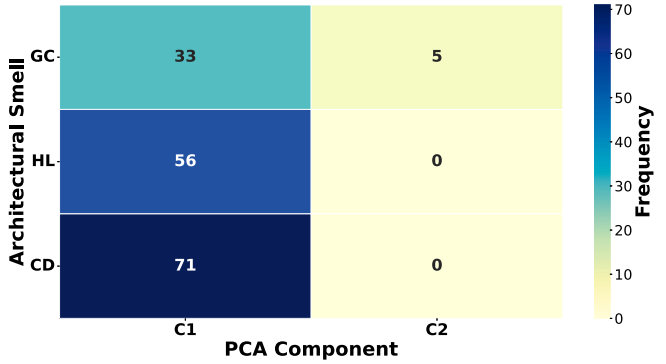
5.2. Answering RQ2

After filtering the matrices, we extracted association rules for 343 Java projects (96% of the initial dataset) and 377 Python projects 88% of the initial dataset). We employed the BCCM matrix, the BPCM matrix, or both for these projects. Tables 5 and 6 concisely summarize the results for Java and Python projects, respectively. The tables report the top 10 rules for each abstraction level and programming language, ordered first by the number of projects, then by conviction, and finally by confidence. This ordering reflects the relative importance assigned to the metrics: the number of projects in which a rule appears is considered the most relevant criterion, as our primary objective is to identify recurring associations across projects. Conviction is also particularly meaningful given the binary presence/absence encoding of each AS type.

The results highlight a consistent tendency of certain AS to co-locate across projects, programming languages, and levels of abstraction. Regarding the rules extracted from Java projects, 54 rules were identified



(a) Class-level AS



(b) Package-level-AS

Fig. 4. Heatmaps showing the frequency of strong associations between AS and PCA components across Python projects ($|loading| \geq .5$).

at the class level and 50 at the package level. At the Java class level, as shown in Table 5, the top 10 Java class-level rules involve four smells: CD, WH, CH, and HL. DH does not appear among the top 10 rules and is present in only 4 of the 54 extracted rules, all of which are confined to a single project. The most frequent associations involve HL and CH, which appear jointly in more than fifty projects and are supported by high confidence values (often above 0.79) and unbounded conviction (∞, no observed counter-examples in the dataset), indicating that the presence of HL smell often implies the presence of the CH. Similarly, the combinations involving CD and WH may confirm the strong interdependence among dependency-related and hierarchy-related issues. At the Java package level, the dominant patterns involve HL and UD. These associations reach the highest number of projects (up to 76) and are characterized by strong confidence values (above 0.69) and unbounded conviction, suggesting a structural interplay between dependency concentration and propagation of instability across packages. There are also extracted rules involving CD, observed in 68 projects with a relatively high average confidence, whereas the GC smell appears in far less frequent rules, with the most common one occurring in only 17 projects.

Turning to the Python projects, 50 rules were extracted at the class level and 12 at the package level. Table 6 shows the top 10 extracted rules. For Python projects at the class level, a different but complementary set of associations emerges. HL is strongly linked to WH, and combinations with CH also occur with high confidence and conviction. The most frequent rules, namely those occurring in more than 50 projects, exhibit a high level of confidence. At the Python package level, the most prominent associations involve GC together with HL and CD. Although these rules occur in fewer projects compared to the Java case, they still display very high confidence and conviction. Therefore, it may appear that excessive component size and a high concentration of dependencies are correlated phenomena.

Table 5
Top 10 association rules extracted from Java projects grouped by abstraction level, ordered by: (1) number of projects, (2) conviction (mean), (3) confidence (mean).

Rule	Projects	Support	Confidence	Conviction
Java class-level AS				
HL ⇒ CH	58	0.0446	0.7902	∞
CH ⇒ HL	58	0.0446	0.0949	1.0421
CD, WH ⇒ CH	53	0.0431	0.9193	∞
CH ⇒ CD, WH	53	0.0431	0.0842	1.0457
CH ⇒ CD	52	0.3248	0.8297	∞
CD ⇒ CH	51	0.3216	0.5379	∞
HL ⇒ CD	46	0.0485	0.7482	∞
CD ⇒ HL	46	0.0485	0.1120	1.0534
CD, HL ⇒ CH	38	0.0360	0.9377	∞
CH, HL ⇒ CD	38	0.0381	0.9127	∞
Java package-level AS				
HL ⇒ UD	76	0.0547	0.6917	∞
UD ⇒ HL	76	0.0547	0.5629	∞
CD, HL ⇒ UD	68	0.0543	0.7036	∞
HL ⇒ CD, UD	68	0.0543	0.7036	∞
CD, UD ⇒ HL	68	0.0543	0.6340	∞
UD ⇒ CD, HL	68	0.0543	0.5970	∞
CD, GC ⇒ HL	17	0.0576	0.7049	∞
GC ⇒ CD, HL	17	0.0576	0.6196	∞
GC ⇒ HL	17	0.0576	0.6196	∞
CD, HL ⇒ GC	17	0.0576	0.6098	∞

It is worth noting, however, that the overall support values remain relatively low, typically below 0.1. This suggests that, although the identified associations are strong when they occur, they are limited to a specific subset of all the components included in the dataset.

When comparing the results with PCA, the findings obtained through ARM appear to confirm the latent patterns identified by PCA for both programming languages. At the class level, a correlation emerges between poorly designed inter-class dependencies, the improper assignment of responsibilities to overly coupled classes, and the disorganization of class hierarchies. In Java, the most recurrent and notable rule highlights the relationship between HL and CH. While these smells also appear in the principal components of the PCA, they are less prominent than CD and WH, likely due to the dataset's imbalance and the lower frequency of HL and CH instances. Since ARM relies on the binary presence variable, it is probably less sensitive to the frequency of instances compared to PCA, which is instead based on the count variable after OLS regression.

At the package level, the results are consistent with those of the PCA. In Java, the most frequent rules (occurring in more than 68 projects) confirm the strong association between cyclic and unstable dependencies and components characterized by an excessive number of dependencies. A similar observation applies to Python, although the available evidence is more limited. This is primarily due to the reduced variety of smells detected at the package level in Python, where Arcan only identifies GC, HL, and CD.

RQ2: Are there systematic co-location patterns among architectural smells that tend to manifest across diverse software projects?

Answer: The ARM revealed that architectural smells tend to co-locate in recurring patterns across many projects. In Java, dependency-related smells (HL, CD, UD) frequently appear together with hierarchy-related ones (CH, WH), while in Python the dominant associations involve HL with WH or CH, and, at the package level, combinations of GC with dependency smells. Overall support is low, meaning these patterns appear in many projects but only within subsets of components.

Table 6
Top 10 association rules extracted from Python projects grouped by abstraction level, ordered by: (1) number of projects, (2) conviction (mean), (3) confidence (mean).

Rule	Projects	Support	Confidence	Conviction
Python class-level AS				
HL ⇒ WH	55	0.0423	0.6186	∞
WH ⇒ HL	55	0.0423	0.3994	∞
CH, WH ⇒ HL	54	0.0379	0.6316	∞
HL ⇒ CH, WH	54	0.0379	0.6018	∞
CH, HL ⇒ WH	53	0.0380	0.8053	∞
WH ⇒ CH, HL	53	0.0380	0.3899	∞
HL, WH ⇒ CH	29	0.0421	1.0000	∞
CH ⇒ HL, WH	29	0.0421	0.0507	1.0096
HL ⇒ CH	28	0.0488	0.9911	∞
CH ⇒ HL	28	0.0488	0.0568	1.0077
Python package-level AS				
CD, HL ⇒ GC	21	0.0889	0.7024	∞
HL ⇒ CD, GC	21	0.0889	0.6786	∞
HL ⇒ GC	21	0.0889	0.6786	∞
CD, GC ⇒ HL	21	0.0889	0.6190	∞
GC ⇒ CD, HL	21	0.0889	0.5635	∞
GC ⇒ HL	21	0.0889	0.5635	∞
HL ⇒ CD	5	0.0877	1.0000	∞
CD ⇒ HL	5	0.0877	0.0972	1.0106
GC, HL ⇒ CD	3	0.0741	1.0000	∞
CD ⇒ GC, HL	3	0.0741	0.0821	1.0089

5.3. Examples of co-location

To illustrate the result of quantitative analysis, we present examples of co-location patterns of AS that emerged from our analyses.

As a first example, we examine a pattern of co-located smells in **Java projects at the class level**. The PCA results reveal a potential latent pattern on the first component, involving the smells WH, CD, and CH. The example comes from the Google Closure Compiler project.[5] Google Closure Compiler processes JavaScript code and transforms it into a more optimized version by analyzing the source, removing unused code, rewriting it, and performing minimization. In the project, we observe that four components exhibit a co-location of the WH, CD, and CH smells. In particular, we note that the component `ObjectType`, located in the package

`com.google.javascript.rhino.jstype`, is affected by 1 CD, 1 WH, and 19 CH instances. The component is an abstract class used, together with the other classes in the same package, to provide abstractions for representing types in JavaScript. Since this class is extended by many classes representing more specific type-related classes, it exhibits a large number of outgoing inheritance dependencies, which explains the presence of the WH smell. All these smells have a severity of 4, thus not particularly high, with the exception of CD, whose severity is 5. The large number of CH instances is due to the fact that the abstract class supertype `JSType` implements several interfaces, which are subsequently implemented by the subtypes of `ObjectType`. The PCA results also suggest a potential pattern involving the same three smells, WH, CD, and CH, together with HL. Within the same project, there is only one component affected by all four smells: it exhibits 1 WH, 1 CD, 83 CH, and 1 HL instances. The component in question is `ParseTree`, a concrete class located in the package

`com.google.javascript.jscomp.parsing.parser.trees`. The severity levels range from 4 to 6, indicating a more critical situation compared to the previous case. In general, the project exhibits a poor design of hierarchies; however, given the overall complexity of the project, performing a comprehensive refactoring would be extremely challenging.

We now provide an example of co-located smells in **Java projects at the package level**. In this case, the results reveal a potential pattern involving the HL, CD, and UD smells. Within the same project mentioned above, we identify two components that are affected by this group of smells. The first example concerns the package `com.google.javascript.jscomp`, which is affected by 1 GC, 1 HL, 1 UD, and 50 CD instances. The GC exhibits a relatively high severity of 7, while both HL and UD have a severity of 6. The CD instances range from a minimum severity of 4 to a maximum of 6. In this case, the package is considerably larger than the other packages at the same hierarchical level (under `com.google.javascript`), and accumulates a substantial number of responsibilities and functionalities, along with a high number of dependencies. However, such design can be interpreted as an intentional architectural choice: this package constitutes the core of the compiler and defines all the public APIs, which justifies its central role and its extensive interactions with other classes. The second example refers to an internal subpackage, `deps`, which contains the classes responsible for mapping dependency information between files. This package is affected by 1 HL, 1 UD, and 11 CD instances. In this case, the package appears to be involved in a potentially excessive number of dependencies, making it "problematic" according to our analyses. The root cause of these issues lies in a suboptimal allocation of responsibilities, which ultimately leads to an ineffective modularization of the project.

We now turn to the examples from the **Python projects**. At the **class level**, the results reveal a potential co-location pattern involving the CH, HL, and WH smells. As an illustrative case, we consider the Connexion project,[6] a web framework written entirely in Python and following the object-oriented paradigm, which aims to simplify and accelerate the development of APIs in standard formats such as OpenAPI [57]. Within the project, we identify the class

`connexion.middleware.abstract.RoutedAPI`, defined in the file `abstract.py`. This class is affected by 7 CH, 1 HL, and 1 WH instances, with severities ranging from 3 to 4, except for the HL instance, which has a severity of 6. In this case, the class serves as a general support component that is specialized and invoked by multiple elements located across different Python packages (similarly to the other classes defined within the same module). This class exhibits a substantial number of dependencies (24) and, as observed by Suryanarayana et al. [5], classes of this type are frequently affected by HL and tend to be coupled to a large number of other classes.

One more example comes from the **Python projects at the package level**. Here, the results indicate a potential pattern involving the GC, HL, and CD smells. However, unlike the other identified patterns, package-level co-location patterns in Python are considerably less frequent. The most common rule discovered through the ARM analysis appears in only 21 projects and relates precisely to these three smells. An example of this co-location pattern can be observed in the BBOT project,[7] which is written entirely in Python and adheres to the object-oriented paradigm. This is an open-source Python framework for Open Source Intelligence (OSINT) and attack-surface reconnaissance. It provides modular scans (such as subdomain enumeration, web crawling, and lightweight vulnerability checks) and is widely used in security analysis and bug-bounty workflows. Within the project, we identify the `bbot.modules` package, which is affected by 56 CD, 1 GC, and 1 HL instances, with severities ranging from 5 to 7, making it a decidedly problematic component. The primary reason is that this package aggregates several submodules and files, leading to an excessive concentration of responsibilities.

Having presented the co-location examples above, we acknowledge that some individual AS may, in some cases, reflect intentional design decisions. However, we argue that, in general, their co-location is less

[5] See https://github.com/google/closure-compiler

[6] See https://github.com/spec-first/connexion

[7] See https://github.com/blacklanternsecurity/bbot.git

likely to represent acceptable architectural trade-offs. The literature provides evidence supporting this view. Sas et al. [58] conclude that co-occurring AS amplify their impact on maintainability and evolvability, giving components more than one reason to change and breaking the Single Responsibility Principle [59]. They also observe that the overlapping of different smell types (such as Hub-like Dependency and Unstable Dependency, or God Component and Cyclic Dependency) creates conditions where changes propagate to parts of the system that are partially or totally unrelated to the original change, a phenomenon directly corroborated by practitioners who experienced such co-locations as sources of severe and unanticipated maintenance burden rather than as intentional design decisions. Furthermore, Sharma et al. [8] demonstrated that design smells (as in our terminology DH, WH, CH, HL) causally precede the emergence of architecture smells in subsequent versions, a pattern of progressive degradation inconsistent with deliberate architectural trade-offs. Finally, Bochicchio et al. [29] show that AS are linked to higher co-change rates in affected files, leading to increased maintenance efforts, therefore suggesting that intentionality does not alter the practical consequences of their presence on maintenance.

6. Discussion and lessons learned

The results of both the PCA and the ARM analyses are consistent and complementary. Overall, the findings reveal patterns emerging in a substantial number of projects, which may indicate co-located smells with a high probability.

At the class level, one notable pattern suggests a structural syndrome of architectural degradation, in which components are simultaneously affected by complex, deeply nested hierarchies (CH and WH) and centralized dependency structures (HL). This co-location may have a significant impact on the maintenance, as the inheritance- and dependency-related negative effects could synergize, leading to quality hotspots in the affected components.

At the package level, another significant pattern emerges, highlighting a syndrome involving components that carry excessive dependency responsibilities (HL) while exhibiting high instability (UD, as measured by Martin's Instability metric [19]), occasionally combined with cyclic dependencies (CD). Among the extracted rules, this co-location (HL and UD) is the most frequently observed across projects (76 Java projects). However, its absolute occurrence count remains relatively low when compared to the entire dataset. A combination of a large number of dependencies with instability will likely produce fragile modules, vulnerable to even more frequent changes.

Based on the analyses conducted, we presented some examples of smell co-location drawn from the projects in our dataset. Taken together, these examples demonstrate that AS often do not occur in isolation. Instead, they could **emerge in clusters** that reflect deeper structural issues, such as misplaced responsibilities, poorly designed hierarchies, or overly tangled dependency structures. The reason for this is that when one design principle is violated (such as one of those defined by Martin [34]), others are likely to be compromised as well, leading to the simultaneous presence of multiple, heterogeneous AS instances.

6.1. Recommendations for practitioners

Based on our results, some important *takeaways* can be highlighted for practitioners.

Patterns of co-located AS statistically tend to appear together, as problematic components frequently violate multiple design principles simultaneously. However, further investigation is required to determine whether these smells actually interact and, if so, how they do so. For example, the co-located AS could pose an even higher risk to the maintenance of the affected components than any of the individual smells separately, amplifying the adverse effects of co-location.

We observe that HL smells frequently appear in the most relevant clusters, often co-located with other smells. This is expected, as HL components are aggregators of functionality, and inadequate separation of responsibilities within them during the design may lead to problematic components and suboptimal hierarchical structures, as reflected in our findings. It supports the observations by Cedrim et al. [60] that the most effective refactorings target aggregator-related smells; we therefore recommend that programmers **carefully monitor and assign higher priority to the refactoring of HL-affected instances**.

Moreover, these clusters often contain one or more instances of Cyclic Dependency smells. CD adversely affects architectural quality, as it violates the Acyclic Dependency Principle [16]. It is essential to **carefully monitor existing CD instances**, as they often co-locate with other smells, may *amplify* their negative impact on the architecture, and foster fragility in the dependency chains.

While those findings require further investigation, they suggest that certain co-located smells may interact, exacerbating maintainability issues not merely through the presence of individual smells, but rather through their co-location and combined effects. This observation aligns with Yamashita's insights regarding code smells [12], indicating that a deeper understanding of smell co-location in the identified patterns could significantly advance both refactoring strategies and prediction tasks.

As systems grow in size and complexity, performing large-scale refactoring becomes increasingly infeasible from a resource and effort standpoint, also challenging their expected maintainability. For this reason, another complementary recommendation for practitioners is to **integrate static analysis tools such as** Arcan, **Designite** [24] **or AI Reviewer**,[8] **into CI/CD pipelines, thus enabling continuous monitoring and management of architectural technical debt**. Moreover, during the design phase, particular attention should be paid to the proper assignment of responsibilities to components, defining well-structured and balanced hierarchies, and using stable dependencies to abstractions rather than to implementations.

6.2. Recommendations for researchers

The results are also important from a research perspective. First, since the co-locations of AS are not random, we may conjecture that they interact, with an adverse impact on the maintainability of the affected components. Although the nature of the effect has not been explored yet, analyzing the evolution of the subject code may contribute to improving our understanding of the characteristics that could be particularly affected by the interaction effects.

Additionally, the prevalence of architectural smells is related to the main relationships between components: inheritance and dependency. Their frequent co-locations highlight the importance of knowledge about, and communication with, other components, in contrast to intra-component issues such as size or complexity, which are more commonly associated with design smells.

Our results for architectural smells indicate that, at the class level, components are often affected simultaneously by complex and deeply nested inheritance hierarchies (CH and WH) and centralized dependency structures (HL). At the package level, components frequently exhibit excessive dependency responsibilities (HL) and high instability (UD), sometimes accompanied by cyclic dependencies (CD). Although smell co-locations at the design and architectural levels cannot be mapped directly, our results suggest that complex dependency structures are the dominant architectural concern. These dependencies are further amplified by inheritance-related problems, instability, or dependency cycles. From this perspective, co-locations of architectural smells are not merely complementary to the design-level smells. The identified AS clusters usually include smells related to inter-class communication

[8] See https://www.aireviewer.com/

and reveal higher-level structural problems that may have a more strategic impact and should therefore be addressed earlier. On the other hand, the design smell co-locations also capture internal class issues related to the size, complexity, or responsibility assignment. This, however, requires investigating the impact of AS co-locations on vital software properties, like defect- or change-proneness.

Moreover, the consistent and non-contradictory results obtained for Java and Python suggest that the co-locations of AS do not result from the idiosyncrasies of the specific language's syntax or its expressiveness, but instead can be attributed to higher-level design principles and their possible violations. This hypothesis, however, should be validated for other programming languages. We also outline that programs in Python, a language rooted in structured design, use object-oriented mechanisms less frequently than the Java ones, and that could affect the distribution of some types of smells. However, even procedural projects do have architecture, and there are AS that could negatively affect it (although these could be different smells and different specific co-locations). In this sense, we broaden the scope of smells and possible co-locations by including Python projects. By comparing Tables 2 and 3, and then Tables 5 and 6, we can see that Java and Python have their own idiosyncrasies, which result in different distributions of smells and different co-locations.

7. Threats to validity

The study presented in this paper is subject to several limitations and threats to its validity. We organize them in terms of *construct validity*, *external validity*, *conclusion validity* and *reliability*. As discussed by Runeson et al. [61], since our work does not investigate causal relationships, *internal validity* is not discussed.

Construct validity. Construct validity is the degree to which we measure exactly what we aim to measure (i.e., software metrics, architectural smells, smell characteristics).

The main threat to construct validity stems from relying solely on Arcan for AS detection, as detection accuracy depends on the tool's implementation. Different technical debt tools often yield divergent results due to varying definitions, assumptions, and detection strategies, limiting comparability with other studies [62]. This threat is partially mitigated because Arcan's AS definitions are grounded in established prior work [16,34,63] and the tool has been evaluated in multiple empirical studies on industrial projects across different programming languages [18,29,33,58].

Furthermore, the absence of UD and CD at the class level in Python projects is not attributable to a limitation of the tool, but rather to the programming paradigm typically adopted in Python projects. As a consequence, the co-location analyses for Python are conducted on a reduced set of AS compared to Java. This is consistently reflected in the results, which are always reported separately by programming language, and does not affect the validity of the findings within the scope of each language.

Another concern pertains to the fact that the results of the statistical tests employed are sensitive to the dataset imbalance. Nevertheless, the dataset can be considered sufficiently representative of reality, as the distribution of smells is consistent with that observed in other datasets reported in the literature [8–10]. In addition, this threat was mitigated by controlling for confounding variables such as code size and Fan-In/Fan-Out metrics.

External validity. External validity concerns the extent to which the results of this study can be generalized to other projects with a similar context. Some threats can be identified.

The external validity is limited by the scope of the dataset. Our results are restricted to two programming languages (Java and Python) and to the set of AS supported by Arcan. Hence, straight extrapolation of results to other languages (e.g., C or C#) or to other AS is not possible. Our conclusions are valid only for the AS and languages considered in the study. This threat is mitigated by the large number of projects analyzed, which increases confidence in the representativeness of the findings within these languages. Furthermore, the languages considered are well-known and widely used.

Conclusion validity. Conclusion validity is the degree to which conclusions about relationships between variables in a study are reasonable, accurate, and supported by data.

In our case, we focus on discovering frequent co-locations of smells. The conclusion is that some smells are statistically more likely to be co-located than others. Threats to the validity are mostly related to (a) the sensitivity/accuracy of the detecting tool (Arcan) with respect to various smells, and (b) the biased selection of subject systems for analysis. For (a), we rely on the previous studies, which found Arcan to be a reliable tool, compared to human perception [22,23,58]. For (b), we justified the selection of systems by applying a stratified sampling with respect to the key project characteristics, including the language, size, and domain.

Reliability. Reliability concerns the extent to which the collected data and the performed analyses depend on the specific researchers involved.

We mitigated reliability threats by making all data, scripts, and results publicly available in a replication package [14]. This allows other researchers to independently reproduce our results, apply different project selection rules, or replicate the analyses with alternative datasets or tools. Additionally, the dataset underwent manual inspection to ensure the appropriateness of the included projects, reducing the potential influence of subjective choices in project selection.

8. Related work

In this section, we provide a brief summary of some empirical studies related to AS and architectural technical debt.

8.1. Impact of AS on code evolution

The studies presented here aim to investigate the impact of AS by examining their evolution.

In the literature, we identified previous empirical studies that investigated the evolution of instability AS, notably the work by Sas et al. [33] and its subsequent extension by Gnoyke et al. [64]. Both studies relied on a dataset of Java projects from the Qualitas Corpus [65] and performed a static analysis of AS using the first academic version of the Arcan tool. Their results revealed that the number of smells does not increase exponentially with system size and that CDs have the most significant impact on architectural degradation. These studies mainly differ in their aims, as they do not investigate the relationships among AS. Moreover, they differ from ours in terms of the dataset size, as our dataset is larger and does not rely on the Qualitas Corpus. Furthermore, they considered only a single programming language (Java), whereas we consider two different languages (Java and Python).

Sas et al. [18] presented an embedded multiple-case study on the evolution and impact of AS in large industrial C/C++ embedded systems. The authors analysed 280 releases from 9 projects to investigate how AS evolve. The authors considered UD, GC at the package level, and HL, CD at both the package- and class level. They investigated the lifecycle of AS, their overlap, and the precedence in which different smells tend to appear. One of their most relevant findings for our study concerns the presence of overlapping smells across system versions. Their results revealed that most smells are likely to overlap with each other, thereby amplifying maintenance issues. While the themes of their work are closely related to ours, there are key differences in the focus and methodology. Specifically, they quantified overlapping instances of smells by analyzing pairwise relationships across multiple system versions. In contrast, our study aims to identify co-location

patterns among multiple types of smells simultaneously, investigating their statistical correlations. Further distinctions include the use of a significantly larger dataset encompassing two programming languages.

In a recent study, Gnoyke et al. [66] performed an empirical study on the evolution of AS. Using an extended version of Arcan and their own tracking tool AsTdEA, they analyzed how intra- and inter-version smells, particularly CD, HL, and UD, emerge, merge, split, and persist over time. The results show that CDs are the most critical AS, as they tend to merge over time and evolve into complex structures, particularly at the class level, making them difficult to refactor. Splits occur less frequently and rarely reduce overall complexity, often leaving the core cyclic structure unresolved. In contrast, instances of UD grow slowly, while the HL ones remain largely stable throughout the system evolution. Unlike our work, the authors considered a much smaller number of projects (14) and focused on a single programming language (Java).

8.2. Correlation between AS and various software characteristics

Other empirical studies, by contrast, have focused on examining the correlation between AS and other aspects of software architecture and maintainability rather than their evolution across multiple versions. For example, Pigazzini et al. [9] investigated the correlation between AS and design patterns. The authors analyzed 60 open-source Java projects using Arcan and *Pattern4* [67] for detecting AS and design patterns, respectively. Although correlation tests revealed no statistically significant relationships, several meaningful associations were identified through ARM. A similarity with our work lies in the use of ARM to investigate correlations, although with a different focus and objectives.

In addition, Jolak et al. [68] conducted an empirical study to investigate the impact of AS on software maintainability, focusing on the quality attributes of modularity and testability. Results showed limited correlation between smells and modularity, except for Dense Structure smell, which exhibited a strong negative effect. Several smells negatively affected testability, though project size emerged as a stronger latent factor.

Other studies, instead, focused on the correlation between AS and code smells, such as the work by Arcelli et al. [10], or on the correlation between AS and the CK quality metrics [69]. There are also other preliminary yet noteworthy studies, such as the one by Herold et al. [70], which investigated the correlation between AS and architectural degradation, and the work by De Stefano et al. [11], which examined the correlation between AS and test smells. Moreover, in a recent study, Gupta [71] investigated the relations between the Temporary Field code smell and 17 design smells (including HL, WH, CH, DH) across 10 popular open-source Java projects, using the TFfinder tool [72] for smell detection. They computed co-occurrence metrics to identify valid relations. The results show that the strongest association is between Temporary Field and Unutilized Abstraction. Moreover, Multifaceted Abstraction always co-occurs with a temporary field. In contrast, temporary field rarely or never co-occurs with several hierarchy- and dependency-related smells.

A relevant empirical study for our work is the one conducted by Sharma et al. [8], which investigates the correlation between AS and design smells. Their detection process relies on the Designite tool [24]. The authors mined 3073 open-source C# repositories (over 118 million LOC), identifying more than 1.2 million instances of seven AS and nineteen design smells. In their study, they also consider smells that are of interest to us, such as CD, UD, and GC, which are classified as AS by the tool, and WH, DH, and CH, which are classified as design smells instead. Their findings indicate that CD, UD, and GC exhibit weak or even no correlation with WH, DH, and CH. Two key differences distinguish our study from that of Sharma et al. First, the use of a different detection tool yields a dataset that is not directly comparable with those produced by alternative tools; moreover, correlation analyses are inherently sensitive to the underlying data distribution. Second, Sharma et al. adopted a different statistical approach, assessing smell-to-smell pairwise relationships by computing Spearman correlation coefficients [73]. Finally, the authors considered only one programming language (C#).

It is worth noting that none of the above studies investigates the correlation among AS.

Finally, several studies have investigated the relationship between AS and co-changes. Sas et al. [40] studied whether the frequency and size of code changes are correlated with the presence of a selected set of AS. The findings, based on a case study of 31 open-source Java systems, show that 87% of the analyzed commits involved changes in artifacts with at least one smell, and the likelihood of change increases with the number of smells. Additionally, there is evidence to suggest that the frequency of change increases after the introduction of a smell and that the size of changes is also larger in smelly artifacts. In another study, Sas et al. [74] investigated co-changes and their relation to AS as proxies of reliability and maintainability. They analysed 14 open-source projects and used two algorithms to detect the Co-changes, which they merged with the AS data to create the dataset. They found that 50% of co-changes participate in an AS, and in 90% of cases, that co-change tends to appear before the introduction of the smell. Moreover, in one of our previous studies, Bochicchio et al. [29] conducted a deeper investigation into the correlation between AS and co-changes, leveraging a data collection pipeline to perform statistical analyses on a large dataset of 118 Java and C# projects. The results showed that file pairs affected by AS are more likely to co-change, and that this likelihood increases with the file size. This led the authors to suggest the following: prioritize refactoring large, smelly files that co-change over clean files, regardless of whether they co-change or not, as breaking the logical coupling between two smelly files is likely to significantly reduce maintenance effort. This previous study is particularly relevant to this work on correlation among AS, as we improved the data collection approach by updating the pipeline and applying more rigorous project selection criteria.

9. Conclusions and future work

In this work, presented a large-scale correlation analysis among AS present in the same software components. This study is novel in that it explores co-locations between AS while also benefiting from the ease and automation provided by our dataset construction process.

The results suggest a potential correlation among co-located AS, where certain patterns of smells appear likely to co-locate within the same components. To complement the statistical analysis, we also provided some illustrative examples of co-located smells. These findings open up promising avenues for further research aimed at determining whether such patterns indeed represent smells that systematically interact with one another.

A potential future research direction would be to replicate the statistical analyses on different programming languages, particularly C# (another fully object-oriented programming language supported by Arcan), or on an alternative dataset of AS detected also by other tools (e.g., Designite [24] or ARCADE [75]). This could enhance the generalizability of the results and increase their robustness. However, the same considerations previously discussed for programming languages also apply here. A further avenue for improvement lies in refining the conducted analyses through the application of additional statistical techniques, such as computing multiple correlation coefficients.

Finally, another interesting research direction is to investigate correlations not only among co-located AS, but also among *coupled* AS. Smells are coupled if they affect coupled components (e.g., packages or classes). Since coupled code smells have been found to have a similar effect on maintainability as the co-located code smells [12], we may expect to observe a similar effect of AS couplings. Identifying patterns among “neighboring” smells could complement the findings and contribute to new insights into the potential interactions between AS.

CRediT authorship contribution statement

Matteo Bochicchio: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Bartosz Walter:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Francesca Arcelli Fontana:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author(s) used ChatGPT in order to assist solely with language-level aspects of the manuscript, such as proofreading, grammar correction, and stylistic refinement. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

Declaration of competing interest

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

Data availability

All the data is publicy available on Zenodo: https://zenodo.org/records/17828125.

References

[1] W. Cunningham, The WyCash portfolio management system, ACM Sigplan Oops Messenger 4 (2) (1992) 29–30.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162), Dagstuhl Rep. 6 (4) (2016) 110–138, http://dx.doi.org/10.4230/DagRep.6.4.110, http://drops.dagstuhl.de/opus/volltexte/2016/6693/ http://drops.dagstuhl.de/opus/volltexte/2016/6693.

[3] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220, http://dx.doi.org/10.1016/j.jss.2014.12.027, http://www.sciencedirect.com/science/article/pii/S0164121214002854.

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

[5] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for software design smells: managing technical debt, Morgan Kaufmann, 2014.

[6] U. Azadi, F.A. Fontana, D. Taibi, Architectural smells detected by tools: a catalogue proposal, in: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE, 2019, pp. 88–97.

[7] R. Verdecchia, I. Malavolta, P. Lago, Architectural Technical Debt Identification: the Research Landscape, in: 2018 ACM/IEEE International Conference on Technical Debt, 2018, pp. 11–20, http://dx.doi.org/10.1145/3194164.3194176.

[8] T. Sharma, P. Singh, D. Spinellis, An empirical investigation on the relationship between design and architecture smells, Empir. Softw. Eng. 25 (5) (2020) 4020–4068, http://dx.doi.org/10.1007/s10664-020-09847-2.

[9] I. Pigazzini, F.A. Fontana, B. Walter, A study on correlations between architectural smells and design patterns, J. Syst. Softw. 178 (2021) 110984, http://dx.doi.org/10.1016/J.JSS.2021.110984.

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

[11] M. De Stefano, F. Pecorelli, D. Di Nucci, A. De Lucia, A preliminary evaluation on the relationship among architectural and test smells, in: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2022, pp. 66–70.

[12] A. Yamashita, M. Zanoni, F.A. Fontana, B. Walter, Inter-smell relations in industrial and open source systems: A replication and comparative analysis, in: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2015, pp. 121–130.

[13] B. Walter, F.A. Fontana, V. Ferme, Code smells and their collocations: A large-scale experiment on open-source systems, J. Syst. Softw. 144 (2018) 1–21.

[14] M. Bochicchio, B. Walter, F. Arcelli Fontana, Dataset and replication package, 2025, http://dx.doi.org/10.5281/zenodo.17828125, Zenodo.

[15] F. Arcelli Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, Proc. - 2016 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2016 (2016) 433–437, http://dx.doi.org/10.1109/ICSME.2016.33.

[16] M. Lippert, S. Roock, Refactorings in large software projects: How to successfully execute complex restructurings, 2006.

[17] J. Garcia, P. Daniel, G. Edwards, N. Medvidovic, Identifying Architectural Bad Smells, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2009, pp. 255–258, http://dx.doi.org/10.1109/CSMR.2009.59.

[18] D. Sas, P. Avgeriou, U. Uyumaz, On the evolution and impact of architectural smells - an industrial case study, Empir. Softw. Eng. 27 (4) (2022) 86, http://dx.doi.org/10.1007/S10664-022-10132-7.

[19] R.C. Martin, Object oriented design quality metrics: An analysis of dependencies, 1995.

[20] D.M. Le, C. Carrillo, R. Capilla, N. Medvidovic, Relating architectural decay and sustainability of software systems, in: Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Institute of Electrical and Electronics Engineers Inc., 2016, pp. 178–181, http://dx.doi.org/10.1109/WICSA.2016.15, URL http://ieeexplore.ieee.org/document/7516827/.

[21] R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells, Proc. - 12th Work. IEEE/IFIP Conf. Softw. Archit. WICSA 2015 (2015) 51–60, http://dx.doi.org/10.1109/WICSA.2015.12.

[22] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, E.D. Nitto, Arcan: A tool for architectural smells detection, Proc. - 2017 IEEE Int. Conf. Softw. Archit. Work. ICSAW 2017: Side Track Proc. (2017) 282–285, http://dx.doi.org/10.1109/ICSAW.2017.16.

[23] F. Arcelli Fontana, F. Locatelli, I. Pigazzini, P. Mereghetti, An architectural smell evaluation in an industrial context, ICSEA 2020 (2020) 68–74.

[24] T. Sharma, P. Mishra, R. Tiwari, Designite: A software design quality assessment tool, in: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, 2016, pp. 1–4.

[25] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, C. Bezerra, How does modern code review impact software design degradation? an in-depth empirical study, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 511–522.

[26] M. Alenezi, M. Zarour, An empirical study of bad smells during software evolution using designite tool, I-Manager's J. Softw. Eng. 12 (4) (2018) 12.

[27] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, A. Oliveira, Removal of design problems through refactorings: Are we looking at the right symptoms? in: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC, IEEE, 2019, pp. 148–153.

[28] T. Sharma, M. Kessentini, Qscored: A large dataset of code smells and quality metrics, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, IEEE, 2021, pp. 590–594.

[29] M. Bochicchio, D. Sas, A. Gilardi, F.A. Fontana, An empirical study on architectural smells through a pipeline for continuous technical debt assessment, Inf. Softw. Technol. (2025) 107783.

[30] M. Raasveldt, H. Mühleisen, Duckdb: an embeddable analytical database, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1981–1984.

[31] R. Van Solingen, V. Basili, G. Caldiera, H.D. Rombach, Goal question metric (gqm) approach, Encycl. Softw. Eng. (2002).

[32] D. Sas, P. Avgeriou, An architectural technical debt index based on machine learning and architectural smells, IEEE Trans. Softw. Eng. 49 (8) (2023) 4169–4195, http://dx.doi.org/10.1109/TSE.2023.3286179.

[33] D. Sas, P. Avgeriou, F. Arcelli Fontana, Investigating instability architectural smells evolution: an exploratory case study, in: 35th International Conference on Software Maintenance and Evolution, IEEE, 2019, pp. 557–567, http://dx.doi.org/10.1109/ICSME.2019.00090, URL https://ieeexplore.ieee.org/document/8919109/.

[34] R.C. Martin, J. Grenning, S. Brown, K. Henney, J. Gorman, Clean Architecture: a Craftsman's Guide to Software Structure and Design, Prentice Hall, 2018.

[35] A.Y. Bahar, S.M. Shorman, M.A. Khder, A.M. Quadir, S.A. Almosawi, Survey on features and comparisons of programming languages (PYTHON, JAVA, AND c#), in: 2022 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems, ICETSIS, IEEE, 2022, pp. 154–163.

[36] H. Borges, A. Hora, M.T. Valente, Understanding the factors that impact the popularity of GitHub repositories, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2016, pp. 334–344.

[37] J. Han, S. Deng, X. Xia, D. Wang, J. Yin, Characterization and prediction of popular projects on github, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference, COMPSAC, 1, IEEE, 2019, pp. 21–26.

[38] K. Boyalakuntla, M. Nagappan, S. Chimalakonda, N. Munaiah, RepoQuester: A tool towards evaluating GitHub projects, in: 2022 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2022, pp. 509–513.

[39] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empir. Softw. Eng. 22 (2017) 3219–3253.

[40] D. Sas, P. Avgeriou, I. Pigazzini, F. Arcelli Fontana, On the relation between architectural smells and source code changes, J. Softw.: Evol. Process. 34 (1) (2021) http://dx.doi.org/10.1002/smr.2398.

[41] A. Ampatzoglou, N. Mittas, A.-A. Tsintzira, A. Ampatzoglou, E.-M. Arvanitou, A. Chatzigeorgiou, P. Avgeriou, L. Angelis, Exploring the relation between technical debt principal and interest: An empirical approach, Inf. Softw. Technol. 128 (2020) 106391.

[42] S. Horschig, T. Mattis, R. Hirschfeld, Do java programmers write better python? studying off-language code quality on github, in: Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming, 2018, pp. 127–134.

[43] F. Arcelli Fontana, V. Ferme, M. Zanoni, A. Yamashita, Automatic metric thresholds derivation for code smell detection, in: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, IEEE, 2015, pp. 44–53.

[44] A. Field, J. Miles, Discovering Statistics Using SAS, sage, 2010.

[45] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: An empirical study, in: 2013 35th International Conference on Software Engineering, ICSE, IEEE, 2013, pp. 682–691.

[46] I.T. Jolliffe, Principal component analysis, second ed., in: Springer Series in Statistics, Springer New York, NY, 2002, http://dx.doi.org/10.1007/b98835.

[47] H.F. Kaiser, The application of electronic computers to factor analysis, Educ. Psychol. Meas. 20 (1) (1960) 141–151.

[48] J. Hair, W. Black, B. Babin, R. Anderson, Multivariate Data Analysis, Cengage, 2019, URL https://books.google.it/books?id=0R9ZswEACAAJ.

[49] F. Acito, Predictive analytics with KNIME, in: Analytics for citizen data scientists, Springer, Switzerland, 2023.

[50] H.F. Kaiser, An index of factorial simplicity, Psychometrika 39 (1) (1974) 31–36.

[51] G.W. Snedecor, W.G. Cochran, Statistical Methods, eighth ed., 1191, (2) Iowa state University press, Ames, Iowa, 1989.

[52] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.

[53] C. Aflori, M. Craus, Grid implementation of the apriori algorithm, Adv. Eng. Softw. 38 (5) (2007) 295–300.

[54] B. Walter, T. Alkhaeir, The relationship between design patterns and code smells: An exploratory study, Inf. Softw. Technol. 74 (2016) 127–142.

[55] D. Juliet Thessalonica, H. Khanna Nehemiah, S. Sreejith, A. Kannan, Intelligent mining of association rules based on nanopatterns for code smells detection, Sci. Program. 2023 (1) (2023) 2973250.

[56] F. Palomba, R. Oliveto, A. De Lucia, Investigating code smell co-occurrences using association rule learning: A replicated study, in: 2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE, 2017, pp. 8–13.

[57] S. Casas, D. Cruz, G. Vidal, M. Constanzo, Uses and applications of the openapi/swagger specification: a systematic mapping of the literature, in: 2021 40th International Conference of the Chilean Computer Science Society, SCCC, IEEE, 2021, pp. 1–8.

[58] D. Sas, I. Pigazzini, P. Avgeriou, F.A. Fontana, The perception of architectural smells in industrial practice, IEEE Softw. 38 (6) (2021) 35–41, http://dx.doi.org/10.1109/MS.2021.3103664.

[59] R. Martin, Agile software development, principles, patterns, and practices. international edition, 2011.

[60] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. De Mello, B. Fonseca, M. Ribeiro, A. Chávez, Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 465–475.

[61] P. Runeson, M. Host, A. Rainer, B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples, John Wiley & Sons, 2012.

[62] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, On the lack of consensus among technical debt detection tools, in: Proceedings - International Conference on Software Engineering, IEEE Computer Society, 2021, pp. 121–130, http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00021, arXiv:2103.04506.

[63] G. Samarthyam, G. Suryanarayana, T. Sharma, Refactoring for software architecture smells, in: Proceedings of the 1st International Workshop on Software Refactoring - IWoR 2016, ACM Press, New York, New York, USA, 2016, pp. 1–4, http://dx.doi.org/10.1145/2975945.2975946, URL http://dl.acm.org/citation.cfm?doid=2975945.2975946.

[64] P. Gnoyke, S. Schulze, J. Krüger, An evolutionary analysis of software-architecture smells, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 413–424.

[65] 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, in: 2010 Asia Pacific Software Engineering Conference, IEEE, 2010, pp. 336–345.

[66] P. Gnoyke, S. Schulze, J. Krüger, Evolution patterns of software-architecture smells: An empirical study of intra-and inter-version smells, J. Syst. Softw. 217 (2024) 112170.

[67] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32 (11) (2006) 896–909.

[68] R. Jolak, S. Karlsson, F. Dobslaw, An empirical investigation of the impact of architectural smells on software maintainability, J. Syst. Softw. 225 (2025) 112382.

[69] I.M.A. Wikantyasa, A.P. Kurniawan, S. Rochimah, CK metric and architecture smells relations: Towards software quality assurance, in: 2023 14th International Conference on Information & Communication Technology and System, ICTS, IEEE, 2023, pp. 13–17.

[70] S. Herold, An initial study on the association between architectural smells and degradation, in: Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14, Springer, 2020, pp. 193–201.

[71] R. Gupta, Discovering smell relations between temporary field and design smells: an empirical analysis, Innov. Syst. Softw. Eng. (2025) 1–22.

[72] R. Gupta, S.K. Singh, Tffinder: A software tool to discover temporary field code smell, in: 2020 2nd International Conference on Advances in Computing, Communication Control and Networking, ICACCCN, IEEE, 2020, pp. 910–914.

[73] L. Myers, M.J. Sirois, Spearman correlation coefficients, differences between, Encycl. Stat. Sci. 12 (2004).

[74] D. Sas, P. Avgeriou, R. Kruizinga, R. Scheedler, Exploring the relation between co-changes and architectural smells, SN Comput. Sci. 2 (1) (2021) 13, http://dx.doi.org/10.1007/S42979-020-00407-5.

[75] D.M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, N. Medvidovic, An empirical study of architectural change in open-source software systems, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 235–245.