Analyzing the Relationship between Community and Design Smells in Open-Source Software Projects: An Empirical Study

Haris Mumtaz
University of Auckland, New Zealand
hmum126@aucklanduni.ac.nz

Paramvir Singh
University of Auckland, New Zealand
p.singh@auckland.ac.nz

Kelly Blincoe
University of Auckland, New Zealand
k.blincoe@auckland.ac.nz

ABSTRACT

Background: Software smells reflect the sub-optimal patterns in the software. In a similar way, community smells consider the sub-optimal patterns in the organizational and social structures of software teams. Related work performed empirical studies to identify the relationship between community smells and software smells at the architecture and code levels. However, how community smells relate with design smells is still unknown.

Aims: In this paper, we empirically investigate the relationship between community smells and design smells during the evolution of software projects.

Method: We apply three statistical methods: correlation, trend, and information gain analysis to empirically examine the relationship between community and design smells in 100 releases of 10 large-scale Apache open-source software projects.

Results: Our results reveal that the relationship between community and design smells varies across the analyzed projects. We find significant correlations and trend similarities for one type of community smell (when developers work in isolation without peer communication—Missing Links) with design smells in most of the analyzed projects. Furthermore, the results of our statistical model disclose that community smells are more relevant for design smells compared to other community-related factors.

Conclusion: Our results find that the relationship of community smells (in particular, the Missing Links smell) exists with design smells. Based on our findings, we discuss specific community smell refactoring techniques that should be done together when refactoring design smells so that the problems associated with the social and technical (design) aspects of the projects can be managed concurrently.

CCS CONCEPTS

• **Software and its engineering → Programming teams**; **Software design engineering**; **Object oriented development**.

KEYWORDS

Community Smells; Social Smells; Design Smells; Socio-Technical Analysis; Open-Source Development.

ACM Reference Format:
Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2022. Analyzing the Relationship between Community and Design Smells in Open-Source Software Projects: An Empirical Study . In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ESEM 2022, Sep. 19–23, 2022, Helsinki, Finland

1 INTRODUCTION

Software development is not an isolated activity; instead it involves a team of managers, stakeholders, and developers [16, 20]. Tasks must be well-coordinated, and communication between the project team members is essential to create a successful product [11–13, 38]. If good communication structure is not present, it may lead to community (or social) smells—sub-optimal patterns in the organizational and social structure of software teams [40]. Such sub-optimal patterns then lead to increased project cost (termed social debt) [39]. Similarly, when software practitioners take poor design decisions during the technical (software) development, the software may contain sub-optimal patterns known as software smells [35]. Software smells can occur at different granularity levels (architecture, design, and source code) and, if not addressed, they can contribute to increased maintenance cost (termed as technical debt) [35].

Conway's law suggests that there is a relationship between the communication structure of a software team and the software design that team creates: *"Organizations, which design systems, are constrained to produce designs which are copies of the communication structures of these organizations."*—Conway's Law [16]. This suggests a relationship between software quality and communication structures, and prior work has shown relationships between developer communication and software quality [13].

Prior work has studied the relationship of community smells with software smells at the architecture-level [37] and the code-level [24, 25]. Palomba et al. [24] found a relationship between code smells and community smells, and suggested that developers contributing to the same code file should communicate about code quality decisions to improve code maintainability and reduce technical debt (i.e., the costs associated with sub-optimal patterns in source code). Similarly, Tamburri et al. [37] found a relationship between the existence of community smells and sub-optimal structures in software architecture (architecture smells). These studies show that if the communication structure is not optimal, it is often associated with maintainability issues in the software [24, 37]. In other words, these studies reported that the relationships of architecture and code smells exist with community smells. However, it is unknown if such a relationship exists between community smells and design smells. Therefore, this study explores the relationship of community smells with design smells. The idea works under the same motivation as previous studies (and Conway's Law [16]) that a software project cannot have an optimal design (i.e., has design

smells) while having a sub-optimal communication structure (i.e., has community smells).

We structure our investigation around the following research question (*RQ*):

RQ *— Is there a relationship between community smells and design smells in software projects?*

This study investigates the relationship between software design smells and software community smells by examining two community smells (Organizational Silo and Missing Links). Both of these smells reflect missing or low communication in the development team [39]. We examine the design smells related to modularization and hierarchy aspects defined by Suryanarayana et al. [36]. The modularization smells include Broken, Insufficient, Hub-like, and Cyclically-dependent, while hierarchy smells under analysis are Wide, Multipath, Cyclic, Rebellious, Missing, and Broken. We collect these community and design smells from 100 releases of 10 large-scale Apache projects (i.e., 10 recent releases from each project, not including patch releases). We investigate the relationship between the community and design smells using three statistical methods: correlation, trend, and information gain analysis. Correlation analysis identifies the association between community and design smells; trend analysis demonstrates the similarities in the trends of community and design smells over releases; finally, information gain analysis explains the extent of the dependency of community smells with design smells by specifying their respective entropy values.

Our results show that the correlations between community and design smells vary across the analyzed projects. The results also reveal that the Missing Links community smell is more significantly correlated and has more significant trend similarities with the design smells under analysis compared to the Organizational Silo community smell. In addition, our statistical model (information gain analysis) reveals that design smells are more dependent on community smells (in particular, Missing Links) in comparison with other community-related factors (socio-technical metrics [13, 23, 40]).

Similar to the previous studies that highlighted the importance of both the community and technical aspects of software development, our findings provide further evidence that, during the evolution of the software, community-aware development is equally important as design development and technical refactoring. Therefore, there is a need to focus on both the design and social aspects while developing a software product.

The main contributions of this study are:

- Empirical evidence of the relationship between community and design smells using a set of 10 large-scale Apache software projects (10 recent releases each—100 releases in total). This empirical study provides further evidence of the importance of both the social and technical aspects of software development. It also provides specific recommendations for ensuring community smells are considered when refactoring to fix design smells.
- A replication package[1] containing: community smells, design smells, socio-technical metrics, and class-level metrics of the 100 releases of the 10 analyzed Apache projects.

[1]https://figshare.com/s/4d2e68474fba8d9ee053

2 MOTIVATING EXAMPLE

In prior work, developers reported that “large teams" are a common cause of communication problems and interaction difficulties, leading to community smells [14]. This same study recommends restructuring the community by splitting large teams into small ones to optimize the communication structure [14]. Inspired by this finding and to motivate the need to understand the relationship between community smells and design smells, we investigate one instance of a “large team” problem. After collecting our data on community and design smells and plotting the evolution of these smells (methods explained in Section 3), we manually examined potential relationships between software design smells and community smells to validate the need for this study prior to performing our empirical analysis. Here, we describe one instance of a large team problem which appeared to be related to a design smell as a motivating example for the empirical study in this paper.

Community smells spiked in the Apache Spark project (an engine for scalable computing) between releases 1.6.0 and 2.0.0 (see Figure 1). Through investigation of communications of the developers between these releases, we observed that several contributors did not participate in the communications related to many classes. Here we discuss one class that was associated with several instances of community smells, `UnsafeInMemorySorter`. In the communication related to this class, we found that in release 2.0.0, half of the contributors did not have any communication at all, causing many instances of community smells. Between the consecutive releases (1.6.0–2.0.0), we also found that the number of contributors working on `UnsafeInMemorySorter` actually doubled (increasing from 4 contributors to 8). This example is in line with the large team communication issues discussed in the literature [14]. We also got interested in the design aspects of `UnsafeInMemorySorter`. Related to design, we observed that the size of `UnsafeInMemorySorter` increased from release 1.6.0 to 2.0.0 because more functionalities were added for release 2.0.0. It is possible that the large team of 8 contributors (working on this single class) had communication difficulties and kept on adding functionalities to `UnsafeInMemorySorter` without consultation with each other, making the class a large one (Insufficient Modularization issue). This example shows the co-occurrence of community smells and a design smell (Insufficient Modularization) in a class.

If a relationship exists between sub-optimal communication patterns (community smells) and sub-optimal design structures (design smells), then refactoring the smells on either side (community or design) becomes important; otherwise, they may create social and technical debt in the projects. This motivated us to deeply investigate the relationship between community and design smells. Inspired by the potential relationships we uncovered through this preliminary manual analysis, we investigate the relationship through an empirical study of 100 releases of 10 open-source software projects (10 recent releases from each project) using statistical methods.

3 METHODS

To analyze the relationship between community and design smells, we collect the data from 100 releases of 10 different Apache open-source projects. To ensure diversity in the selected projects, each

project was selected from a different category (defined by Apache[2]) to have a representation from different application domains. All selected projects were categorized under the Java language.[3] To ensure all projects in our dataset are good-sized and have active communities, we used the sampling protocol and thresholds employed by Palomba et al. [24]; we selected projects that have considerable codebase size (at least 100 classes per release), longevity (at least 5 years long), activity (more than 1000 commits), and developers (more than 20). Table 1 summarises the details of the 10 Apache projects under analysis. In our analyzed projects, the minimum and maximum classes in a release are 109 and 6324, respectively.

3.1 Smells

3.1.1 Community Smells. In this study, we investigate the communication structure of the analyzed projects using two community smells: Organizational Silo and Missing Links, defined as:

- *Organizational Silo*. When there are isolated sub-groups of developers that do not communicate except through one or two of their members [39]. This smell counts the number of collaboration edges (analyzed in groups of two contributors) in which one of the co-committing contributors does not communicate at all.
- *Missing Links*. When developers work in isolation without communicating with their peers [39]. This smell counts the number of collaboration edges that do not have a communication counterpart.

The rationale for selecting these community smells is their frequent occurrence in the community structure of open-source software projects [40]. Furthermore, they have shown correlations with other community-related health indicators [40]. We leave analysis of other community smells for future work.

3.1.2 Design Smells. Previous studies used architecture and code smells to investigate the structural issues in the architecture and source code, respectively [24, 37]. In this study, we consider the structural issues in software design that are reflected by modularization and hierarchy smells. The rationale for studying these smells is their ability to reflect the modularization and hierarchical issues in the design of software projects, which can lead to maintainability issues (an important quality aspect for software evolution) [35]. We adopt the modularization and hierarchy smells from Suryanarayana et al. [36]. We leave an analysis of other types of design smells for future work. We analyze all modularization smells defined by Suryanarayana et al. [36]:

- *Broken Modularization*. When classes are modularized too much that they have only a few data members and methods (that often show interest in other classes) [36].
- *Insufficient Modularization*. Classes with many data members and methods or classes with a few methods with excessive implementation [36].
- *Hub-like Modularization*. Classes with a lot of dependencies with other classes [36].
- *Cyclically-dependent Modularization*. Classes depending on each other forming a cycle of interactions [36].

These modularization smells consider the design entities (classes) that are either too big or too small; or have dependency issues. The modularization issues could potentially relate with community smells. For instance, developers working in isolation could end up creating many small classes in the design with a high number of dependencies.

We analyze a subset of hierarchy smells (defined by Suryanarayana et al. [36]) because we could not find instances of some of the hierarchy smells (i.e., Unnecessary, Unfactored, Speculative, and Deep) in the projects analyzed in this study. Hierarchy smells under analysis are:

- *Wide Hierarchy*. When intermediate types are missing from the hierarchy, the hierarchy may become wide [36].
- *Multipath Hierarchy*. When subtype inherits both directly and indirectly from a supertype [36].
- *Cyclic Hierarchy*. When a supertype has a reference of its subtype, it introduces a cycle [36].
- *Rebellious Hierarchy*. When supertype and subtype share a “is-a" relationship; however, some methods violate this relationship [36].
- *Missing Hierarchy*. When conditional logic is used to manage different behaviors instead of using hierarchical structure [36].
- *Broken Hierarchy*. When supertype and subtype semantically do not share a “is-a" relationship; however, such relationship has been created [36].

These hierarchy smells mainly reflect the design structures that have relationship issues (e.g., hierarchical relationships are not implemented, or, if implemented, they are incorrect). The hierarchy smells could potentially relate with community smells. For instance, developers working in isolation could not be aware of similar classes that could benefit from a hierarchy structure.

3.1.3 Smells Collection. We employed the tools *Designite*[4] [33] and *Kaiaulu*[5] [26] to collect the design and community smells, respectively. To collect the design smells, we downloaded the project's source code and provided this as input to the the *Designite* tool. To collect the community smells, we took the following steps:

(1) Prepared configuration files of the analyzed projects by providing all the necessary paths (e.g., communication channels' paths, gitlog path) and other required information, such as projects' release dates and time slice (in days) between two releases.
(2) Collected communication data from commonly used communication channels: Apache Mailing List, Jira, and GitHub using *Kaiaulu*.
(3) Calculated community smells for the analyzed projects using *Kaiaulu*.

3.2 Analysis

We employed three statistical methods to analyze the relationship between community and design smells: correlation analysis, trend analysis, and information gain analysis. Correlation analysis explains the overall strength of the relationship. To see whether the

[2]https://www.apache.org/#by-category
[3]https://projects.apache.org/projects.html?language
[4]https://www.designite-tools.com/designitejava/
[5]https://github.com/sailuh/kaiaulu/

Table 1: Apache projects under analysis

Project	Category	Commits	Devs	Duration	Releases
Ant [1]	Build management	≈15k	62	22 years	1.1, 1.2, 1.3, 1.4, 1.5, 1.6.0, 1.7.0, 1.8.0, 1.9.0, 1.10.0
Cassandra [2]	Database	≈27k	353	12 years	3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10
Jackrabbit [5]	Network-server	≈9k	26	16 years	2.12.0, 2.13.0, 2.14.0, 2.15.0, 2.16.0, 2.17.0, 2.18.0, 2.19.0, 2.20.0, 2.21.0
Jena [6]	Library	≈9.7k	78	10 years	3.9.0, 3.10.0, 3.11.0, 3.12.0, 3.13.0, 3.14.0, 3.15.0, 3.16.0, 3.17.0, 4.0.0
JMeter [7]	Testing	≈17.5k	47	22 years	3.0, 3.1, 3.2, 3.3, 4.0, 5.0, 5.1, 5.2, 5.3, 5.4
Karaf [8]	OSGI	≈8.9k	145	14 years	2.0.0, 2.1.0, 2.2.0, 2.3.0, 3.0.0, 2.4.0, 4.0.0, 4.1.0, 4.2.0, 4.3.0
Spark [10]	Big data	≈32.4k	1,786	12 years	1.3.0, 1.4.0, 1.5.0, 1.6.0, 2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.4.0, 3.0.0
CloudStack [3]	Cloud	≈34.9k	352	11 years	4.6.0, 4.7.0, 4.8.0, 4.9.0, 4.10.0, 4.11.0, 4.12.0, 4.13.0, 4.14.0, 4.15.0
CXF [4]	Network-client	≈16.7k	172	14 years	2.3.0, 2.4.0, 2.5.0, 2.6.0, 2.7.0, 3.0.0, 3.1.0, 3.2.0, 3.3.0, 3.4.0
Nutch [9]	Web framework	≈3.2k	46	12 years	2.3, 1.10, 1.12, 1.13, 1.14, 1.15, 2.4, 1.16, 1.17, 1.18

smells (community and design) evolve over releases in a similar manner or not, we applied trend analysis (specifically, *Mann-Kendall test*). Lastly, we employed information gain analysis because it can quantify the actual gain provided by the variables of interest in the model by ranking them based on the information gain provided.

3.2.1 Correlation Analysis. Correlation explains the degree of the relationship between variables [27]. We employed Spearman's correlation to assess the association of community and design smells. We computed the Spearman's correlation because the data is not normally distributed [27]. We interpreted the correlations as follows: 0–0.19 as *negligible*, 0.20–0.29 as *weak*, 0.30–0.39 as *moderate*, 0.40–0.69 as *strong*, and equal or greater than 0.70 as *very strong* [17].

3.2.2 Trend Analysis. We employed the *Mann-Kendall test* because it can explain whether the variables of interest (i.e., community and design smells) follow a similar upward or downward trend (or no trend at all) as they evolve over releases. The null hypothesis of the *Mann-Kendall test* represents that there is no trend, whereas the alternate hypothesis means that a trend exists (either upward or downward). The coefficient value of the test indicates the strength of the similarity between the variables (i.e., higher coefficient values mean strong similarities). Therefore, we exploit the *Mann-Kendall test* to examine the similarity in the trends of the design and community smells over releases. Given the number of releases (10) for each project, if the trend analysis shows similar trends of community and design smells over time, it is an indication that there could be a relationship between the smells.

3.2.3 Information Gain Analysis. Information gain analysis explains the extent of the relationship between independent and dependent variables by quantifying the gain provided by each variable in the model [28]. Since we are interested in identifying the extent of the relationship of community smells with design smells, we chose this statistical modeling by estimating the mutual information gain of each variable [21]. Similar to Palomba et al. [24], we treated design smells as dependent variables and community smells as independent variables. We also included various community-related and non-community-related (technical) control factors in the gain analysis (statistical modeling). We employed similar control factors in the statistical model, presented by Palomba et al. [24, 25], which examined the relationship between community smells and code smells.

We considered the following community-related control factors (calculated using the *Kaiaulu* tool) based on the rationale that they are indicators of the project's social health [23, 40]:

- *Socio-technical Congruence.* It is the degree of agreement between the communication needs of software projects and the actual communications that occur within a software development environment [13, 23, 40]. Socio-technical congruence measures the direct comparison of the collaborations (representing all the development relationships and communication needs) to the communications (representing all the actual coordination relationships within a software development team) [13, 23, 40].
- *Code.only.devs.* This metric counts the number of contributors (developers) in the collaboration network who do not participate in the communication channels [23, 40].
- *Code.files.* We use the number of classes (*code.files*) that were changed between two releases. The intuition is that the greater the number of modified classes (i.e., more collaborations), the higher the likelihood that communication issues may appear [23, 40].

From the design perspective, we also control for non-community-related factors (computed using the *Designite* tool) that indicate the project's technical health because they are related to software design structure. Size, coupling, complexity, and inheritance are common maintainability quality attributes [29]. Therefore, we use the following well-known and empirically-evaluated, object-oriented metrics estimating size, coupling, complexity, and inheritance as control factors:

- *Lines of Code (LOC).* It counts the lines of code in the class (size) [15].
- *Coupling between Objects (CBO).* The number of classes that a class references (coupling) [15].
- *Weighted Method per Class (WMC).* It measures the sum of complexities of methods of a class (complexity) [15].
- *Depth of Inheritance (DIT).* It aggregates the classes that a particular class inherits from (inheritance) [15].

The output of the gain analysis is an ordered (descending) selection of metrics based on their degree of dependency (given by the entropy values—higher value means more dependency). The most relevant metric for the dependent variable is placed at the

top followed by the metrics in the order of their relevance to the dependent variable.

4 RESULTS

In this section, we present the statistical analysis results explaining the relationship between community and design smells.

4.1 Correlation Analysis

The results in Table 2 and Table 3 show that the correlations of community and design smells vary across projects (i.e., some projects show correlations and some do not). We find that more projects demonstrate significant correlations of design smells with Missing Links compared to Organizational Silo. For instance, most of the time, at most 4 projects (out of 10) have shown significant correlations of design smells with Organizational Silo. On the other hand, more projects (on average 6 projects) demonstrate significantly strong correlations between design smells and Missing Links. Only one design smell, Multipath Hierarchy, has more significant correlations with Organizational Silo compared to Missing Links; however, this is significant for only 2 of the 10 projects (Ant and Karaf). There are also instances in Table 2 and Table 3 where correlations are zero because specific design smells did not change over releases in the projects, yielding no correlations.

4.2 Trend Analysis

Overall, the trend similarities of design smells are more evident with Missing Links than Organizational Silo in the analyzed projects. In terms of modularization smells, we find that more projects have similar trends with Missing Links, except in the case of Broken Modularization, where the number of projects demonstrating similar trends is the same (7 projects each) for both of the community smells—see Table 4. From the hierarchy smells perspective, we find that more projects have trend similarities in four hierarchy smells (Wide, Cyclic, Missing, and Broken) with Missing Links; while, for the remaining two hierarchy smells (Multipath and Rebellious), the number of projects showing the similar trends with both the community smells is same (6 projects each)—see Table 4. It can be observed that mostly the trends are either upward or no-trend except in a couple of instances in design smells where the trends are downward (i.e., Rebellious Hierarchy smell in Apache Jena and Wide Hierarchy smell in Apache CXF—highlighted in bold in Table 4). These downward trends happen because instances of these smells were removed from the design as these projects (Jena and CXF) evolved.

The examples of Apache Spark (Figure 1) and Ant (Figure 2) projects show the evolution of design (modularization and hierarchy) and community smells over releases. These figures illustrate the similarities in the trends of design and community smells. Both design and community smells spike in the same releases; in addition, they touch their peaks at the same time. For instance, between releases 1.6.0 and 2.0.0 of Apache Spark, community smells increase sharply when there is also a sharp increase in design smells. In another instance in Apache Spark, during another spike from 2.4.0 to 3.0.0 in community smells, we also observe the same sharp increase in modularization and hierarchy smells (see Figure 1). Moreover, during the periods when the increase in community smells is subtle, we also observe the similar slight upward trend in design smells. For instance, in Figure 1, the upward slope of Missing Links from release 2.1.0 to 2.4.0 resembles that of the Broken Hierarchy trend (2.1.0–2.4.0). Similarly, in the Apache Ant project, both community and design smells reach their peaks in release 1.7.0 (see Figure 2). Both design and community smells follow an upward trend until release 1.7.0, with most of the smells either dropping or staying constant thereafter (see Figure 2). All of these trend similarities in the examples of Apache Spark and Ant projects (also in other projects as indicated by the *Mann-Kenadall* test in Table 4) show that the design and community smells under study behave almost the same way as they evolve over releases (i.e., exhibiting their relationship temporally).

4.3 Information Gain Analysis

Table 5 and Table 6 report the results of the information gain analysis by listing the community smells and control factors (in descending order) in accordance with their dependency with design smells. From the community point of view, we find that all of the analyzed design smells have shown some level of dependency with the considered community smells (see Table 5 and Table 6). In addition, we find that community smells can explain the relationship with design smells better than the other socio-technical metrics do. For instance, for Insufficient Modularization, Missing Links has significantly higher entropy reduction (0.41) in comparison to socio-technical congruence (0.02). There are a couple of exceptions, e.g., in the cases of Hub-like Modularization and Wide Hierarchy, where community-related factors (code.only.devs and socio-technical congruence, respectively) can explain the dependency with design smells better than community smells (See Table 5 and Table 6). In addition, both of the community smells have shown relevance for different design smells. It can be seen in Table 5 that Organizational Silo has demonstrated relationships with Cyclically-dependent Modularization, Multipath Hierarchy, and Cyclic Hierarchy; whereas Missing Links is more relevant for the Broken Modularization, Insufficient Modularization, Rebellious Hierarchy, Missing Hierarchy, and Broken Hierarchy smells.

The results also show that design smells are most dependent on the non-community-related (technical) control factors (LOC, CBO, WMC, and DIT). The most likely reason for this behavior is because these non-community-related control factors are generally used to detect the analyzed design smells [36]; therefore, this relationship is natural and expected. Among the non-community-related control factors, coupling is the most relevant maintainability factor because Coupling between Objects (CBO) has shown the highest gain for most of the analyzed design smells (see Table 5 and Table 6).

Answer to *RQ* — We have identified relationships between the community smells and the design smells under study. The Missing Links smell has demonstrated more significant correlations and trend similarities with the design smells in the analyzed projects. Community smells (especially, Missing Links) are more relevant than the other community-related metrics (e.g., socio-technical congruence) in explaining the relationship with design smells.

Table 2: Spearman's correlation of modularization smells with community smells

	Broken		Insufficient		Hub-like		Cyclically-dependent	
Project	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**
Ant	0.78**	0.74*	0.81**	0.78**	0.74*	0.74*	0.94***	0.89***
Cassandra	0.14	0.71*	0.5	0.76*	0.28	0.75*	0.2	0.47
Jackrabbit	0.74*	0.64*	0.22	0.34	0.74*	0.64*	0.31	0.21
Jena	0.25	-0.02	0.29	-0.08	-0.25	-0.55	0.61*	0.41
JMeter	-0.04	-0.26	0.32	0.5	0.35	0.53	0.56	0.7*
Karaf	0.74*	0.71*	0.77**	0.81**	0.00	0.00	0.75*	0.76*
Spark	0.62*	0.7*	0.78**	0.78**	0.9***	0.78**	0.79**	0.77**
CloudStack	-0.44	-0.65*	0.44	0.63*	0.41	0.41	0.34	0.58
CXF	0.01	0.76*	0.22	0.73*	0.12	0.78**	-0.13	0.73*
Nutch	0.09	-0.09	-0.03	-0.14	0.00	0.00	0.07	-0.17

O_S is Organizational Silo and **M_L** is Missing Links

* p <0.05, ** p <0.01, *** p <0.001

0–0.19: *negligible*; 0.20–0.29: *weak*; 0.30–0.39: *moderate*; 0.40–0.69: *strong*; and ≥ 0.70: *very strong*

No correlation (0.00) occurs when the design smell remains constant over releases, hence, zero correlation

Table 3: Spearman's correlation of hierarchy smells with community smells

	Wide		Multipath		Cyclic		Rebellious		Missing		Broken	
Project	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**	**O_S**	**M_L**
Ant	0.85**	0.79**	0.86**	0.81**	0.77**	0.72*	0.92***	0.87**	0.59	0.65*	0.89***	0.83**
Cassandra	0.54	0.84**	0.53	0.29	0.56	0.78**	0.81**	0.94***	0.66*	0.91***	0.51	0.76*
Jackrabbit	0.74*	0.64*	0.00	0.00	0.74*	0.64*	0.74*	0.64*	0.25	0.32	-0.12	0.13
Jena	0.73*	0.4	-0.14	-0.43	0.31	0.17	-0.32	-0.07	0.47	0.47	0.22	-0.04
JMeter	0.45	0.64*	0.04	0.26	0.45	0.64*	-0.4	-0.35	0.00	0.00	0.44	0.67*
Karaf	0.6*	0.69*	0.65*	0.57	0.65*	0.65*	0.00	0.00	0.00	0.00	0.8**	0.79**
Spark	0.67*	0.67*	0.52	0.52	0.76*	0.61*	0.9***	0.78**	0.85**	0.77**	0.8**	0.79**
CloudStack	0.65*	0.65*	0.3	0.53	0.00	0.00	0.00	0.21	0.00	0.00	0.58	0.75*
CXF	-0.11	-0.82**	0.29	0.51	-0.01	0.18	0.31	0.69*	0.21	0.86**	0.22	0.73*
Nutch	0.00	0.00	0.00	0.00	0.04	-0.03	0.00	0.00	-0.1	-0.21	0.24	-0.02

O_S is Organizational Silo and **M_L** is Missing Links

* p <0.05, ** p <0.01, *** p <0.001

0–0.19: *negligible*; 0.20–0.29: *weak*; 0.30–0.39: *moderate*; 0.40–0.69: *strong*; and ≥ 0.70: *very strong*

No correlation (0.00) occurs when the design smell remains constant over releases, hence, zero correlation

5 DISCUSSION AND VALIDITY THREATS

In this section, we first discuss our results and afterward present the validity threats of this study.

5.1 Discussion

Community-aware development is also important alongside technical development. In this study, community smells have shown their relevance in explaining the presence of design smells. In addition, the correlations and similarities in the evolution of design and community smells suggest that they have a strong relationship. Previous studies also discovered the importance of removing community smells because they could contribute to the intensity of code and architecture smells [24, 37]. This study provides further evidence that social aspects should also be given attention when practitioners highlight the importance of quality during technical development. While the information gain analysis of this study found that non-community (technical) factors are the most relevant ones for design smells, this should be interpreted with caution since these metrics are used within the design smell detection methods.

Literature has separately discussed in detail how the refactoring of the design [36] and community [14] smells should be approached. Catolino et al. [14] discussed several refactoring strategies that can be applied to remove Organizational Silo and Missing Links community smells. Suryanarayana et al. [36] presented various refactoring methods for removing the modularization and hierarchy design smells. However, we believe it is important that the refactoring of these community and design smells is accomplished together to tackle the community and technical issues in the projects at

Table 4: Mann-Kendall trend analysis of design and community smells

Smell	Ant	Cassandra	Jackrabbit	Jena	JMeter	Karaf	Spark	CloudStack	CXF	Nutch
Broken Modularization	29[↑]*	[–]	[–]	24[↑]*	[–]	33[↑]**	17[↑]*	[–]	35[↑]**	26[↑]*
Insufficient Modularization	45[↑]***	42[↑]***	29[↑]**	28[↑]*	33[↑]**	45[↑]***	45[↑]***	43[↑]***	45[↑]***	[–]
Hub-like Modularization	32[↑]**	27[↑]*	[–]	[–]	[–]	[–]	29[↑]**	[–]	40[↑]***	[–]
Cyclically-dependent Modularization	38[↑]***	29[↑]**	[–]	[–]	[–]	41[↑]***	42[↑]***	41[↑]***	34[↑]**	29[↑]*
Wide Hierarchy	35[↑]**	29[↑]**	[–]	[–]	33[↑]**	37[↑]***	23[↑]*	[–]	**22[↓]***	[–]
Multipath Hierarchy	33[↑]**	[–]	[–]	24[↑]*	[–]	24[↑]*	[–]	35[↑]**	31[↑]**	[–]
Cyclic Hierarchy	39[↑]***	41[↑]***	[–]	[–]	33[↑]**	21[↑]*	27[↑]*	[–]	27[↑]*	[–]
Rebellious Hierarchy	28[↑]*	[–]	[–]	**-24[↓]***	[–]	[–]	29[↑]**	24[↑]*	39[↑]***	[–]
Missing Hierarchy	24[↑]*	25[↑]*	[–]	[–]	[–]	[–]	32[↑]**	[–]	24[↑]*	[–]
Broken Hierarchy	33[↑]**	44[↑]***	35[↑]**	[–]	37[↑]***	41[↑]***	43[↑]***	37[↑]**	45[↑]***	27[↑]**
Organizational Silo	30[↑]**	[–]	[–]	[–]	[–]	28[↑]*	27[↑]*	[–]	[–]	[–]
Missing Links	28[↑]*	25[↑]*	[–]	[–]	[–]	31[↑]**	31[↑]**	[–]	23[↑]*	[–]

Community smells are in *italic*

* p <0.05, ** p <0.01, *** p <0.001

The higher the coefficient value, the stronger the trend similarity; [↑] = Increasing trend; [↓] = Decreasing trend; [–] = No trend

Table 5: Information gain analysis of modularization smells

Broken		Insufficient		Hub-like		Cyclically-dependent	
Variable	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**
CBO	1.59	CBO	1.43	CBO	1.38	WMC	1.80
LOC	1.55	LOC	1.28	DIT	1.27	LOC	1.73
DIT	1.44	DIT	1.03	LOC	0.88	CBO	1.70
WMC	1.29	WMC	0.92	WMC	0.86	DIT	1.48
Missing Links	0.56	*Missing Links*	0.41	Code.only.devs	0.39	*Org. Silo*	0.92
Code.only.devs	0.54	*Org. Silo*	0.26	*Org. Silo*	0.37	*Missing Links*	0.69
ST Cong.	0.45	Code.only.devs	0.23	*Missing Links*	0.36	ST Cong.	0.57
Org. Silo	0.38	Code.files	0.07	ST Cong.	0.29	Code.only.devs	0.45
Code.files	0.36	ST Cong.	0.02	Code.files	0.21	Code.files	0.39

Community smells are in *italic*

Higher gain value means more dependency of the variable on the design smell

Table 6: Information gain analysis of hierarchy smells

Wide		Multipath		Cyclic		Rebellious		Missing		Broken	
Variable	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**	**Variable**	**Gain**
DIT	1.34	CBO	1.31	CBO	1.05	CBO	1.57	CBO	1.90	DIT	1.83
CBO	1.28	DIT	1.25	WMC	0.89	DIT	1.30	DIT	1.57	CBO	1.71
LOC	1.19	LOC	1.24	DIT	0.86	WMC	1.10	LOC	1.47	LOC	1.42
WMC	1.09	WMC	1.07	LOC	0.70	LOC	1.07	WMC	1.38	WMC	1.13
ST Cong.	0.53	*Org. Silo*	0.38	*Org. Silo*	0.45	*Missing Links*	0.65	*Missing Links*	0.73	*Missing Links*	0.93
Missing Links	0.45	*Missing Links*	0.34	*Missing Links*	0.26	Code.only.devs	0.64	Code.only.devs	0.65	*Org. Silo*	0.67
Code.only.devs	0.37	Code.only.devs	0.23	Code.only.devs	0.22	*Org. Silo*	0.57	*Org. Silo*	0.59	Code.only.devs	0.63
Org. Silo	0.34	ST Cong.	0.16	Code.files	0.14	Code.files	0.36	ST Cong.	0.50	Code.files	0.29
Code.files	0.33	Code.files	0.08	ST Cong.	0.14	ST Cong.	0.36	Code.files	0.33	ST Cong.	0.13

Community smells are in *italic*

Higher gain value means more dependency of the variable on the design smell

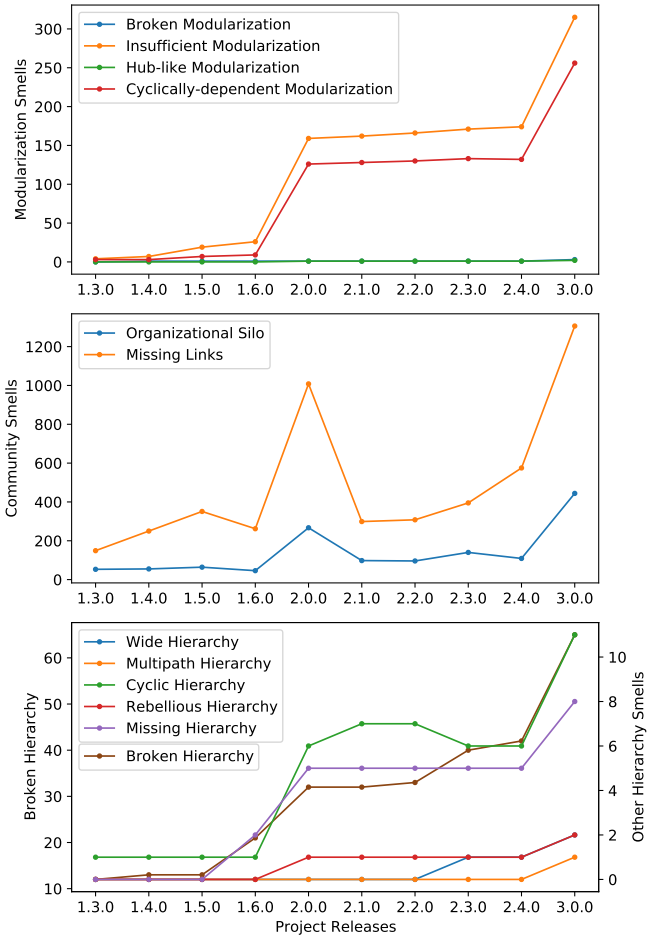

Figure 1: Line plots of design and community smells in Apache Spark. Broken Hierarchy and other hierarchy smells are plotted on two different y-axes because of the differences in their scales.

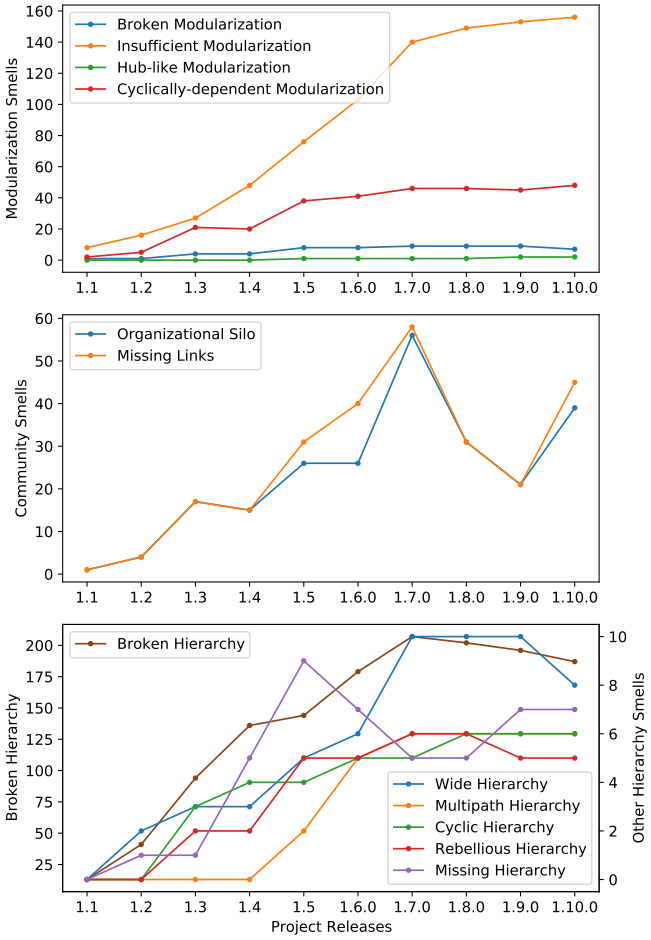

Figure 2: Line plots of design and community smells in Apache Ant. Broken Hierarchy and other hierarchy smells are plotted on two different y-axes because of the differences in their scales.

the same time. If community and design smells are not refactored together, the smells are likely to reemerge because of their relationships.

For instance, we observed that, in release 1.5.0 of Apache Ant, five classes (`TreeBasedTask`, `StarTeamCheckin`, `StarTeamCheckout`, `StarTeamList`, and `StarTeamLabel`) implemented semantically incorrect "is-a" relationships (i.e., Broken Hierarchy instances). From release 1.5.0 to release 1.7.0, no refactoring was performed to remove the Broken Hierarchy issues in these five classes. In addition, the community smells also escalated between these releases (1.5.0–1.7.0)—see Figure 2. However, in release 1.8.0, both the community smells and Broken Hierarchy instances decreased (see Figure 2). The decline in Organizational Silo and Missing Links indicates the removal of the community issues in the project. On the design side, we found that developers removed the semantically incorrect "is-a" relationships (i.e., Broken Hierarchy instances) associated with `TreeBasedTask`, `StarTeamCheckin`, `StarTeamCheckout`, `StarTeamList`, and `StarTeamLabel`, improving the hierarchical structure of the project. This example scenario demonstrates the simultaneous increase of the community smells and a design smell as the project evolved, and the possible collective refactorings of the smells that lead to the improvement in both the community and design structures of the Apache Ant project. Future work could perform in-depth qualitative analysis of such cases to better understand these relationships.

In the remainder of the discussion, we suggest how the separate refactoring methods of the community and design smells (presented by Catolino et al [14] and Suryanarayana et al. [36], respectively) can be joined to eradicate social and technical issues together.

Missing Links has demonstrated relationships with the design structures that are either not implemented at all (i.e., missing) or implemented when not required (i.e., broken). For instance, both Insufficient Modularization and Missing Hierarchy reflect the design structures that are not implemented by

the developers (i.e., classes are not modularized and do not have a desired hierarchical structure). We found that if modularization is missing or hierarchy is missing, they have strong relationships with the Missing Links community smell. This means that the developers who are working on the project without coordinating with their peers (i.e., Missing Links) can create classes that are either too big (i.e., Insufficient Modularization) or do not create hierarchies at all, even when required (i.e., Missing Hierarchy). Similarly, such developers can also create classes that are too small (i.e., Broken Modularization) or create hierarchical relationships that are semantically incorrect (Broken Hierarchy).

For refactoring Insufficient Modularization, large classes are split into multiple maintainable classes [36]. Team restructuring is a common refactoring strategy to improve the communication in the community. Therefore, large teams (working on large classes) can be split into smaller ones as the large classes are split into multiple smaller classes. Broken Modularization can be removed by moving methods between classes to reduce dependencies [36]. Similarly, on the social side, team restructuring will move the developers to work on classes where methods are moved.

To eradicate Broken Hierarchy issues, the semantically incorrect relationships ("is-a") should be removed [36]. To ensure that such incorrect relationships are not created in the future, software teams should perform communication mentoring (defined as teaching all team members how to best share information [14]) and cohesion exercises (defined as activities to improve team cohesion [14]) to identify issues in the communication structure and improve cohesion between the team members so that Broken Hierarchy issues can be discussed proactively. Similarly, Missing Hierarchy can be refactored by replacing behaviors (conditional logic) with hierarchies [36]. On the social side, team restructuring can create dedicated teams for implementing each behavior as a hierarchy; furthermore, communication mentoring will ensure that incorrect hierarchies are not created in the design.

Design smells that focus on cycles between the classes (e.g., Cyclically-dependent Modularization and Cyclic Hierarchy) have a relationship with Organizational Silo. This means that missing communications because of the disjoint sub-communities in the organizational structure (Organizational Silo) are related to the sub-optimal ways classes communicate with each other (i.e., classes communicating with each other in cycles). Such cycles in the design should be removed by moving methods so that dependencies no longer exist [36]. On the community side, the team's communication issues can be removed by restructuring the team through functionality re-assignments to the classes where methods are moved. Additionally, the software teams can create communication protocols in advance so that teams are well-aware of how the communication should be executed, ensuring cyclic structures are not introduced in the future design.

The two design smells (Hub-like Modularization and Wide Hierarchy) that are more related with socio-technical factors are similar in a way that both create a hub-like design structure (i.e., one main class and with many associated classes). In Hub-like Modularization, there is often a central class and many classes communicate with it; whereas, in Wide Hierarchy, there is a root class with many directly derived classes [36]. Although we saw that these design smells had correlations and trend similarities with the community smells in the analyzed projects, in information gain analysis, both showed more relevancy with the socio-technical metrics. Specifically, Hub-like Modularization showed dependency with "code.only.devs", while "socio-technical congruence" is more relevant for Wide Hierarchy. Both these socio-technical metrics reflect that the social side of the projects is not optimal; thus, we see two sub-optimal patterns (Hub-like Modularization and Wide Hierarchy) in software design. The refactoring of these design smells mainly requires moving methods between the classes; therefore, for refactoring community structure, the community side of the projects should perform team restructuring and define communication protocols so that these design smells can be removed efficiently.

In our discussion, we presented several refactoring suggestions to deal with social and technical quality issues in the projects jointly. For all of these suggestions, it may not always be possible to restructure the software teams, so in these cases the design refactoring should carefully consider the existing team structures. Future work can perform experiments to validate these suggestions.

5.2 Threats to Validity

We explain the validity threats of our empirical study in terms of four categories (as presented by Wohlin et al. [43]).

5.2.1 Conclusion Validity. The conclusion validity refers to the ability to draw correct conclusions regarding the relationship between the dependent and independent variables [43]. In our empirical analysis, imprecise measurement of the variables could be a conclusion validity threat because error-in-variable bias could occur if variables are measured imprecisely, leading to incorrect conclusions. To mitigate this threat, we employed two empirically-validated and widely-used tools (*Designite* [30, 31] and *Kaiaulu* [18, 41]) to automate the collection of design and community smells, respectively. We also captured the communication data (required for computing social smells) automatically using the *Kaiaulu* tool. Another conclusion validity threat in our study is the threshold-based design smell detection employed the *Designite* tool, which can potentially introduce bias, particularly when the detected (or undetected) smells are in close proximity to the predefined thresholds in the tool. However, *Designite* has been empirically-validated [30, 31], and is the current state-of-the-art design smell detection tool.

5.2.2 Construct Validity. The construct validity deals with the accurate representation of the theoretical concepts in the dependent and independent variables [43]. Design and community smells are indications of problems in the software designs and communities, respectively; however, they may not always reflect true problems. To mitigate this, we used a set of well-known, frequently-occurring, and validated design and community smells. However, other types of design and community smells may yield different or additional insights on their relationship in open-source projects.

5.2.3 Internal Validity. Threats to internal validity capture concerns where factors that affect the dependent variables have not been accounted for [43]. We collected communication data from three commonly used channels (Apache Mailing Lists, Jira, and GitHub); still, we cannot guarantee the completeness of the communication data as there could be other communication channels used in the analyzed projects. In addition, we adopted several control

factors to mitigate the problem of wrong interpretations because of missing confounding factors that can also be related with design smells. We employed both community-related and non-community-related factors to cover both social and technical aspects. Similar control factors were used in the related studies. However, we cannot be sure that we have not missed other important confounding factors; particularly since relationships between smells varied across projects. Future work can include other confounding factors that may yield further insights into the relationships under analysis.

5.2.4 External Validity. The validity threat is related to the generalization of the experimental results [43]. To mitigate this, we considered 100 releases of 10 large-scale Apache open-source projects which each have a good quantity of development and communication activities. We ensured the projects came from different domains. However, we cannot claim the results generalize beyond our dataset. Replicating this study with additional open-source projects may produce more insights into our results. Furthermore, we used projects from the Apache organization that are categorized under Java development. To further validate our findings, future work can select projects developed in different languages and from other organizations (having different development and communication norms). Finally, our results are limited to the smells analyzed in this study. Future work can investigate a wider range of smells.

6 RELATED WORK

Here, we discuss the research related to the relationships between different types of software smells (architecture, design, and code), and then we examine the literature that has investigated the relationship of community smells with these software smells.

6.1 Relationship between Software Smells

Sharma et al. [34] empirically investigated the relationship between design and architecture smells using correlation, collocation, and causation analysis. They found that architecture smells are strongly correlated with design smells. Similarly, Fontana et al. [19] conducted an empirical study to investigate the relationship between three architecture smells and many code smells. However, they found that architecture smells do not influence the existence of code smells. Palomba et al. [22] and Walter et al. [42] explored the collocation of design and code smells using open-source software projects. Both of these studies reported the co-occurrence of smells in the analyzed projects. In an empirical study, Sharma et al. [32] explored inter- and intra-category relationships between design and code smells and reported the existence of their association.

These studies did not examine community smells and focused only on the technical aspects. In contrast, the study described in this paper considered the relationship between community smells and design smells. For examining the technical aspects of software projects, we used design smells as adopted by Sharma et al. [32, 34].

6.2 Relationship between Community and Software Smells

Only a few studies explored the relationship between community and software smells. Tamburri et al. [37] investigated the relationship of architectural smells with community smells using correlation analysis in agile teams. They reported a strong correlation between community smells and architecture smells. Palomba et al. [24, 25] reported that community smells contribute to the intensity of code smells and are often correlated. Palomba et al. [24] first surveyed the developers to understand if the decision of refactoring the code smells is influenced by the community-related issues (i.e., community smells). Alongside other factors, developers highlighted many community smells that influence the decision of refactoring code smells. Furthermore, Palomba et al. [24, 25] applied information gain analysis (statistical modeling) to see the extent to which community smells explain the code smells intensity. Their model used common object-oriented metrics (e.g., LOC, CBO) and community smells (including socio-technical metrics) as independent variables. They reported that object-oriented metrics are the most powerful predictors for code smells. Furthermore, they concluded that community smells influence the intensity of code smells. For instance, missing communications (Organizational Silo) can misplace the classes, or Missing Links can influence the evolution of methods (Feature Envy smell) [24]. Lastly, they found that socio-technical metrics can explain code smells intensity; however, community smells showed better dependency than socio-technical metrics.

These studies investigated the relationships of community smells with architecture and code smells. However, we examined the relationship between community smells and design smells, which has not yet been explored. Our study employed the same statistical methods (e.g., correlation and information gain analysis), as applied in the related works. Furthermore, we collected communication data from two additional channels (Jira and GitHub) compared to the related studies where communication data is based only on Apache Mailing Lists.

7 CONCLUSION

In this paper, we examined the relationship between community and design smells in 100 releases of 10 large-scale open-source Apache projects using three methods: correlation analysis, trend analysis, and information gain analysis. Our results reported that correlations between community and design smells vary across the analyzed projects. The Missing Links community smell showed more significant correlations and trend similarities with the design smells compared to Organizational Silo community smell. In information gain analysis, design smells showed dependency on community smells more so than socio-technical metrics (e.g., socio-technical congruence). The relationships between design smells and community smells (especially, Missing Links) provide further evidence that equal focus should be given to technical (design) and community aspects during software development. This is important because software is developed by a community, and if the community does not have effective communication, software design quality can suffer. In our discussion, we suggest specific joint refactoring strategies to ensure that the community and design smells are fixed together. We suggest future work to conduct experiments to validate these joint refactoring strategies so that their practical usefulness can be highlighted. Future work can also survey developers to better understand how software practitioners deal with the social and technical issues during the evolution of software projects and validate the collective refactorings of community and design smells discussed in our work.

REFERENCES

[1] Apache. 2022. *Ant.* https://projects.apache.org/project.html?ant
[2] Apache. 2022. *Cassandra.* https://projects.apache.org/project.html?cassandra
[3] Apache. 2022. *CloudStack.* https://projects.apache.org/project.html?cloudstack
[4] Apache. 2022. *CXF.* https://projects.apache.org/project.html?cxf
[5] Apache. 2022. *Jackrabbit.* https://projects.apache.org/project.html?jackrabbit
[6] Apache. 2022. *Jena.* https://projects.apache.org/project.html?jena
[7] Apache. 2022. *JMeter.* https://projects.apache.org/project.html?jmeter
[8] Apache. 2022. *Karaf.* https://projects.apache.org/project.html?karaf
[9] Apache. 2022. *Nutch.* https://projects.apache.org/project.html?nutch
[10] Apache. 2022. *Spark.* https://projects.apache.org/project.html?spark
[11] Kelly Blincoe, Giuseppe Valetto, and Daniela Damian. 2015. Facilitating Coordination between Software Developers: A Study and Techniques for Timely and Efficient Recommendations. *IEEE Transactions on Software Engineering* 41, 10 (2015), 969–985. https://doi.org/10.1109/TSE.2015.2431680
[12] Marcelo Cataldo and James D Herbsleb. 2012. Coordination Breakdowns and their Impact on Development Productivity and Software Failures. *IEEE Transactions on Software Engineering* 39, 3 (2012), 343–360. https://doi.org/10.1109/TSE.2012.32
[13] Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. 2008. Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the 2nd ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2–11. https://doi.org/10.1145/1414004.1414008
[14] Gemma Catolino, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Filomena Ferrucci. 2020. Refactoring Community Smells in the Wild: The Practitioner's Field Manual. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*. 25–34. https://doi.org/10.1145/3377815.3381380
[15] Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895
[16] Melvin E Conway. 1968. How Do Committees Invent. *Datamation* 14, 4 (1968), 28–31.
[17] Christine P Dancey and John Reidy. 2007. *Statistics without Maths for Psychology.* Pearson Education.
[18] Manuel De Stefano, Emanuele Iannone, Fabiano Pecorelli, and Damian Andrew Tamburri. 2022. Impacts of Software Community Patterns on Process and Product: An Empirical Study. *Science of Computer Programming* 214 (2022), 102731. https://doi.org/10.1016/j.scico.2021.102731
[19] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. 2019. Are Architectural Smells Independent from Code Smells? An Empirical Study. *Journal of Systems and Software* 154 (2019), 139–156. https://doi.org/10.1016/j.jss.2019.04.066
[20] James Herbsleb and Jeff Roberts. 2006. Collaboration in Software Engineering Projects: A Theory of Coordination. (2006).
[21] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating Mutual Information. *Physical Review E* 69, 6 (2004), 066138. https://doi.org/10.1103/PhysRevE.69.066138
[22] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A Large-scale Empirical Study on the Lifecycle of Code Smell Co-occurrences. *Information and Software Technology* 99 (2018), 1–10. https://doi.org/10.1016/j.infsof.2018.02.004
[23] Fabio Palomba and Damian Andrew Tamburri. 2021. Predicting the Emergence of Community Smells using Socio-Technical Metrics: A Machine-learning Approach. *Journal of Systems and Software* 171 (2021), 110847. https://doi.org/10.1016/j.jss.2020.110847
[24] Fabio Palomba, Damian Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells? *IEEE transactions on software engineering* 47, 1 (2018), 108–129. https://doi.org/10.1109/TSE.2018.2883603
[25] Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, Andy Zaidman, Francesca Arcelli Fontana, and Rocco Oliveto. 2018. How Do Community Smells Influence Code Smells?. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion*. IEEE, 240–241.
[26] Carlos Paradis and Rick Kazman. 2021. Design Choices in Building an MSR Tool: The Case of Kaiaulu. In *1st International Workshop on Mining Software Repositories for Software Architecture*.
[27] Susan Prion and Katie Anne Haerling. 2014. Making Sense of Methods and Measurement: Spearman-Rho Ranked-Order Correlation Coefficient. *Clinical Simulation in Nursing* 10, 10 (2014), 535–536. https://doi.org/10.1016/j.ecns.2014.07.005
[28] J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine learning* 1, 1 (1986), 81–106.
[29] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. 2009. A Systematic Review of Software Maintainability Prediction and Metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 367–377. https://doi.org/10.1109/ESEM.2009.5314233
[30] Tushar Sharma. 2017. Designite: A Customizable Tool for Smell Mining in C# Repositories. In *10th Seminar on Advanced Techniques and Tools for Software Evolution, Madrid, Spain*.
[31] Tushar Sharma. 2019. How Deep is the Mud: Fathoming Architecture Technical Debt using Designite. In *2019 IEEE/ACM International Conference on Technical Debt*. IEEE, 59–60. https://doi.org/10.1109/TechDebt.2019.00018
[32] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2017. House of Cards: Code Smells in Open-source C# Repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 424–429. https://doi.org/10.1109/ESEM.2017.57
[33] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: A Software Design Quality Assessment Tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*. 1–4.
[34] Tushar Sharma, Paramvir Singh, and Diomidis Spinellis. 2020. An Empirical Investigation on the Relationship between Design and Architecture Smells. *Empirical Software Engineering* 25, 5 (2020), 4020–4068. https://doi.org/10.1007/s10664-020-09847-2
[35] Tushar Sharma and Diomidis Spinellis. 2018. A Survey on Software Smells. *Journal of Systems and Software* 138 (2018), 158–173. https://doi.org/10.1016/j.jss.2017.12.034
[36] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt.* Morgan Kaufmann.
[37] Damian Tamburri, Rick Kazman, and Willem-Jan Van den Heuvel. 2019. Splicing Community and Software Architecture Smells in Agile Teams: An Industrial Study. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.
[38] Damian A Tamburri, Rick Kazman, and Hamed Fahimi. 2016. The Architect's Role in Community Shepherding. *IEEE Software* 33, 6 (2016), 70–79. https://doi.org/10.1109/MS.2016.144
[39] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. 2015. Social Debt in Software Engineering: Insights from Industry. *Journal of Internet Services and Applications* 6, 1 (2015), 1–17. https://doi.org/10.1186/s13174-015-0024-6
[40] Damian Andrew Andrew Tamburri, Fabio Palomba, and Rick Kazman. 2019. Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2901490
[41] Jari van Meijel. 2021. On the Relations Between Community Patterns and Smells in Open-Source: A Taxonomic and Empirical Analysis. (2021).
[42] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. 2018. Code Smells and their Collocations: A Large-scale Experiment on Open-source Systems. *Journal of Systems and Software* 144 (2018), 1–21. https://doi.org/10.1016/j.jss.2018.05.057
[43] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2003. Empirical Research Methods in Software Engineering. In *Empirical Methods and Studies in Software Engineering*. Springer, 7–23. https://doi.org/10.1007/978-3-540-45143-3_2