



An empirical investigation on the relationship between design and architecture smells

Tushar Sharma¹ · Paramvir Singh² · Diomidis Spinellis³

Published online: 28 August 2020

© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

Context: Architecture of a software system represents the key design decisions and therefore its quality plays an important role to keep the software maintainable. Code smells are indicators of quality issues in a software system and are classified based on their granularity, scope, and impact. Despite a plethora of existing work on smells, a detailed exploration of architecture smells, their characteristics, and their relationships with smells in other granularities is missing.

Objective: The paper aims to study architecture smells characteristics, investigate correlation, collocation, and causation relationships between architecture and design smells.

Method: We implement smell detection support for seven architecture smells. We mine 3 073 open-source repositories containing more than 118 million lines of C# code and empirically investigate the relationships between seven architecture and 19 design smells.

Results: We find that smell density does not depend on repository size. Cumulatively, architecture smells are highly correlated with design smells. Our collocation analysis finds that the majority of design and architecture smell pairs do not exhibit collocation. Finally, our causality analysis reveals that design smells cause architecture smells.

Keywords Maintainability · Code quality · Code smells · Architecture smells · Design smells · Inter-smell relationships · Collocation · Correlation · Causality · Refactoring

Communicated by: Barbora Buhnova

✉ Tushar Sharma
tusharsharma@ieee.org

Paramvir Singh
p.singh@auckland.ac.nz

Diomidis Spinellis
dds@hua.gr

¹ Siemens Corporate Technology, Charlotte, NC, USA

² School of Computer Science, University of Auckland, Auckland, New Zealand

³ Athens University of Economics and Business, Athens, Greece

1 Introduction

In the software development lifecycle, software design spans from low-level class design to high-level architecture design. The quality of software design, including all its facets, plays an important role in keeping a software system maintainable.

Code smells (Fowler 1999; Sharma and Spinellis 2018b) are indicators of software quality deterioration that make software hard to maintain. Based on granularity, scope, and impact, smells are classified as implementation (Fowler 1999), design (Suryanarayana et al. 2014), and architecture smells (Garcia et al. 2009). Implementation smells are confined to a limited scope, typically to a method, and require relatively less effort to refactor. Design smells have a larger scope and impact and therefore refactoring a design smell may introduce a change in a set of classes. The architecture of a software system represents the critical design decisions that span multiple components and have a system-level impact (Bass et al. 2012). Thus, architecture smells affect a set of components and require considerable effort to refactor (Martini et al. 2018) given their relatively wider scope. Although the effort to refactor smells matches the increase in their granularity, the corresponding benefits and positive consequences also increase significantly (Samarthyam et al. 2016).

Over the fifteen years, the software engineering community has proposed many code smell detection mechanisms (Sharma and Spinellis 2018b). These are divided into categories such as metric-based (Marinescu 2004), rule-based (Moha et al. 2010), history-based (Palomba et al. 2013), and machine learning-based (Arcelli Fontana et al. 2016) methods. Among these categories, metric-based mechanisms are the most widely employed. However, the existing research mainly supports implementation and some design smells. The research on architecture smells and their detection is still in a budding stage (Garcia et al. 2009; Samarthyam et al. 2016), and requires serious attention from the community given their importance and impact on the quality of software systems.

There have been some attempts to understand the impact of code smells on architecture quality. For instance, Macia et al. (2012a) investigated the impact of nine code smells on five architecture smells and revealed that code smells are related to 78% architectural anomalies in the studied systems. Also, they found that some code smells show a higher correlation with architecture smells. Some authors (Oizumi et al. 2014; Macia et al. 2012a, b) studied the interplay among smells at different granularities and deduced that the inter-related smells affect architecture quality negatively. Source code granularity has been found to be helpful in similar contexts such as architecture recovery (Lutellier et al. 2018) and defect prediction (Cui et al. 2019).

Despite these attempts to understand the impact of code smells on architecture as well as leveraging the inter-smell relationships to find optimal sequences to refactor smells (Liu et al. 2012), the relationships (such as correlation and collocation) among smells at various granularities have not been explored in detail. *Collocation* refers to a relation between a smell pair instance that is detected in the same source code element (such as a class) (Yamashita et al. 2015; Walter et al. 2018) whereas *correlation* measures the degree of co-occurrence in terms of number of instances detected in a source code element (such as a component) between two kinds of smells (Fontana et al. 2019; Sharma et al. 2016a). Related work (Oizumi et al. 2014; Macia et al. 2012a, b) suggests that there could be a *causal* relationship among smells at design and architecture smells; however, to the best of our knowledge, none of the earlier work attempts to establish causality between design and architecture smells. In addition, this work attempts to broaden the scope of the analysis both in terms of the number of smells detected and analyzed as well as the number of analyzed repositories.

This paper contributes to the current research pertaining to the field of architecture smells in the following ways:

- *Correlation and collocation analysis between code smell categories:* We analyze a large set of repositories and infer relationships among the smells belonging to two granularities (architecture and design) through correlation and collocation analysis. In particular, the analysis explores whether there are specific design smells that may act as indicators for specific architecture smells. This exploration may reveal the intricate relationships among smells at different granularities and help us glean insights from the presence or absence of explored relationships.
- *Causality analysis between design and architecture smells:* We investigate temporal relationship between smell types to figure out whether design (or architecture) smells cause architecture (or design) smells.
- *Automated architecture smell detection:* We automate the detection of a set of seven architecture smells.
- *Code smell dataset:* We build a large dataset comprising 1 232 348 instances of seven architecture and 19 design smells, extracted from 3 073 open-source C# repositories. The dataset is available online ([Sharma 2019a](#)) for the use of software engineering research community.

The rest of the paper is structured as follows. Section 2 explains the motivation behind this work. Then we discuss the existing literature in Section 3. Section 4 outlines theoretical background describing all the design and architecture smells considered in this work along with their detection mechanism. We define and briefly explain the research questions undertaken in Section 5. We also describe the experimental study design including selection of subject systems and the overall experimental method. Results of the experiments and observations corresponding to each research question are presented in Section 6. We extend our discussion on the explored relationships along with implications to software development community in Section 7. Finally, we conclude the paper with some promising future work in Section 9.

2 Motivation

The software engineering community has defined several code smells belonging to different granularities ([Sharma and Spinellis 2018b](#)). Many of these smells seem to be related to each other following different kinds of relationships—some smells show presence of other smells while some smells lead to other smells. [Yamashita et al. \(2015\)](#) and [Mantyla et al. \(2003a\)](#) explore some of the relationships. In this section, we discuss a motivating example to establish a theoretical relationship between *unutilized abstraction* ([Suryanarayana et al. 2014](#)) design smell and *feature concentration* ([de Andrade et al. 2014](#)) architecture smell. This relationship helps in forming the basis of our research presented in this paper.

Unutilized abstraction design smell arises when an abstraction (e.g. class) is not used at all by other abstractions in a software system ([Suryanarayana et al. 2014](#)). *Feature concentration* architecture smell occurs when a software component implements more than one architectural feature or concern ([de Andrade et al. 2014](#)). This smell is detected by computing *Lack of Component Cohesion* (LCC) metric. LCC is computed by first identifying and grouping the related classes (by association, aggregation, composition, or inheritance) of a component in the form of a dependency graph. Dividing the number of disconnected subgraphs by the total number of classes in the component gives us the value of the metric.

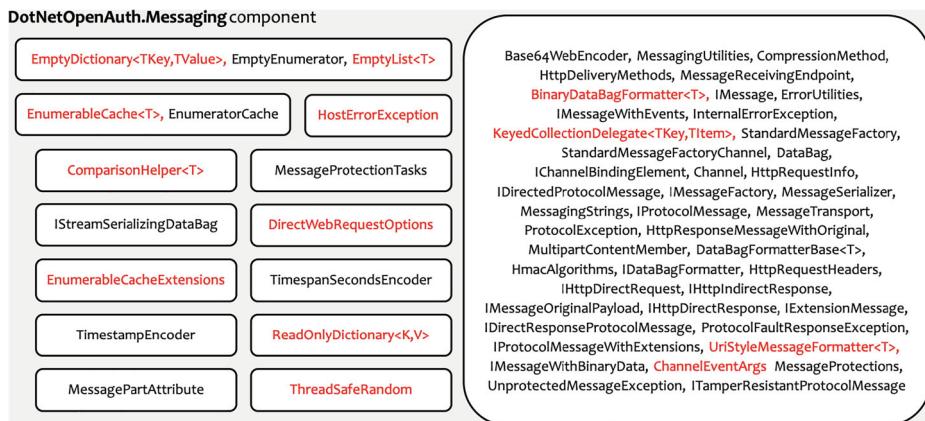


Fig. 1 Motivational example—Component *DotNetOpenAuth.Messaging* has 60 classes; each box represents a disconnected sub-graph and contains related classes (via association, aggregation, composition, or inheritance). Class names written in red color show *unutilized abstractions*

A component with LCC greater than a pre-defined threshold is identified as the component suffering from *feature concentration* smell (refer to Section 4.2 for detection mechanism of *feature concentration* smell).

Figure 1 shows all 60 classes belonging to *DotNetOpenAuth.Messaging* component of *DotNetOpenAuth.Core* project in an open-source C# repository¹. We form sub-graphs of the related classes as described above and put them in a box; therefore, each box represents a disconnected sub-graph.

The grouping of classes results in 14 sub-graphs. The computed LCC is $14/60 = 0.21$ that makes the component suffering from *feature concentration* smell (assuming the threshold of $LCC = 0.20$).²

Interestingly, 13 classes are not used anywhere (shown in red color in Fig. 1) and hence suffer from *unutilized abstraction* smell. Now, it can be observed that the reason for six sub-graphs to be detected as disconnected is rooted in the *unutilized abstraction* design smell instances (since all the classes in these sub-graphs are unutilized). These instances are contributing to the increase in the LCC value, and are, in turn introducing the *feature concentration* smell in the component. If we refactor each of these *unutilized abstraction* smell instances, the number of disconnected subgraphs reduces to eight and the total number of classes to 47. The new LCC value after the refactoring is $8/47 = 0.17$. This implies that the component no longer suffers from the *feature concentration* architecture smell. It can, therefore, be expected that a higher number of unutilized abstractions present at the design granularity in a component increases the chances of *feature concentration* smell at the architecture granularity. It also implies that if we refactor *unutilized abstraction* smell from a component, it might lead to the removal of *feature concentration* architecture smell.

Identifying architecture smells and removing them is an important yet daunting task (Samarthyam et al. 2016). Software development teams often hesitate to refactor

¹<https://github.com/DotNetOpenAuth/DotNetOpenAuth>

²In cases where there is no previous work recommending specific threshold values, we chose a value based on our experience in analyzing source code and after experimenting with various threshold values and manually analyzing the resultant set of detected smells.

architecture smells given their larger scope and risks (Samarthyam et al. 2016). For example, refactoring a *god component* architecture smell may result in creating another component with relevant classes from the god component. This refactoring may break the build unless all the references to classes moved to the new component are fixed accordingly. Furthermore, if some of the components are owned by different geographically dispersed teams, the need for refactoring has to be communicated to the teams and a consensus has to be built before actually carrying it out. We believe that exploring the relationships among architecture smells and smells at finer-granularity (such as design granularity) may facilitate easier adoption of architecture refactoring. This will help researchers and practitioners in understanding the role of finer-granularity code smells in introducing various architecture smells, in turn cautioning developers against the architecture degradation abilities of such smells.

3 Related Work

The topic of architecture smells and its impact on software development has been a subject of interest for software engineering community for many years. We divide the literature related to this paper into four major parts: architecture smells, smells relationships, the impact of code smells on architecture degradation, and smell detection tools.

3.1 Architecture Smells

Garcia et al. (2009) define an initial set of architecture smells including *ambiguous interface* and *scattered parasitic functionality*. Garcia (2014) provides mathematically formal definitions of smells that help implement the smell detection tools for architecture smells. Duc Minh (Le et al. 2016) presents an extended catalog of architecture smells along with the impacted quality attributes. A number of attempts (Mantyla et al. 2003b; Pietrzak and Walter 2006; Lanza et al. 2005; Fontana et al. 2015) have been made to deliberate on introducing conceptual definitions along with the categorizations of smell relationships.

Similarly, Brown et al. (1998) documents a set of architecture smells, among organization and process antipatterns, in enterprise settings. de Andrade et al. (2014) define a set of architecture smells for Product Line Architecture (PLA). Tamburri et al. (2019) explore the impact of community smells on architecture quality. An open-source architecture repair model is defined by Tran et al. (2000). They argue that the difference between the conceptual architecture and concrete architecture is the main reason behind the occurrence of architecture smells. Behnamghader et al. (2017) utilize ARCADE to compute change and decay metrics associated with architecture evolution.

Recently, some focused research attempts have been made to understand a specific kind of technical debt (Kruchten et al. 2012) *i.e.*, *architecture technical debt* (Besker et al. 2018). In particular, architecture debt identification has attracted considerable effort (Verdeccchia et al. 2018; Li et al. 2015). Along the same lines, Mo et al. (2019) explores the impact of refactoring on architecture debt. Similarly, Koziolek et al. (2013) present an industrial case study to ensure the sustainability of software architecture using architecture metrics.

3.2 Smell Relationships

The focus of the current research that investigates the relationships among code smells has been mainly on either design or implementation granularity. The current research on smell relationships hardly involves architecture smells. Recent years have witnessed a number

of empirical studies (Lozano et al. 2015; Yamashita and Moonen 2013a) that explore inter-smell relations. Yamashita et al. (2013b, 2015) empirically investigate the inter-smell relationships, termed as *collocated* and *coupled*, among 24 and 12 code smells, respectively. They explore mainly design smells and a few implementation smells. Their study reveals that the explanatory power of code smell relationships needs further investigations with complementary perspectives in order to be deemed useful. Similarly, Ma et al. (2015) identify several coupled and some conflicting smell pairs among a set of 10 code smells.

Palomba et al. (2018) investigate the collocation (termed as co-occurrences) among 13 design and implementation smells over multiple releases of 30 open-source software systems. In a parallel work, Walter et al. (2018) conduct an experimental study to explore the collocation relationship among 14 design and implementation smells across 92 Java applications. They employ a variety of smell detection tools to explore the impact of tool selection on the smell relationships. Additionally, they explore the effect of the application domain on these relationships. Both of these studies foresee the importance of smell collocations in identifying classes requiring high maintenance effort and developing appropriate refactoring approaches as well as support from tools to detect smell collocation.

Fontana et al. (2019) carry out an empirical study on the relationship between architecture and code smells. They analyze the correlation between a set of 3 architecture smells (*viz.* *unstable dependency*, *hub-like dependency*, and *cyclic dependency*) and 19 code smells including both implementation and design smells. They use Arcan and SonarQube tools to detect architecture and code smells in 111 Java subject systems. They report that the presence of architecture smells do not depend on the presence of code smells.

Sharma et al. (2017) conduct a large-scale mining study to understand the relationships between design and implementation smells. Specifically, the study explores inter-category and intra-category correlation between smells belonging to different granularities. Bavota et al. (2015) mine code smells in 3 Java-based programs and explore the relationship of smells with refactoring and software quality. Similarly, a study by Fontana et al. (2013) reveals characteristics of smells such as the frequently occurring smells in various application domains.

3.3 Impact of code smells on architecture

It is believed that the code smells affect software architecture in a negative way. Macia et al. (2012b) present an empirical study on the relationship between code anomalies and architecture degradation. They reveal the deficiency of the present set of smell detection tools to capture the correlated architecture problems.

Oizumi et al. (2015, 2016) believe that the architecture problems are reflected in the source code through groups of code smells and study the impact of a number of code smell agglomerations on architecture problems. Guimaraes et al. (2014, 2018) conduct a controlled experiment utilizing architecture blueprints to prioritize various types of code smells based on their architectural relevance.

Martini et al. (2018) conduct a case study on three architectural smells employing questionnaires, interviews, and code inspections on four industrial software projects. The main aim of the study is to identify and prioritize the architecture debt with the help of architecture smells. The findings of the study acknowledge the adverse effects of architecture smells and emphasize on the unavailability of automatic smell detection tools. Le et al. (2018) perform an empirical investigation on the nature and impact of six architecture smells that most frequently appeared in a set of eight Apache Software Foundation open-source projects. They designed detection algorithms for these smells and explored relationships between

issues and architecture smells under study. The outcome of the study states the negative impacts of architecture smells on maintenance effort in terms of the increased number of implementation issues and code commits.

3.4 Smell Detection Tools

A plethora of smell detection tools has been developed by the community using various techniques. The prominent techniques used to detect smells are metrics-based (Marinescu 2004; Santiago et al. 2016a, b), rules-based (or heuristics) (Moha et al. 2010), machine learning-based (Khomh et al. 2009), and history-based (Palomba et al. 2015). However, most of the present tools support the detection of a very scanty kind of smells and target mainly Java programming language. Moreover, the existing tools mainly focus on implementation and a limited set of design smells.

There have been some attempts to detect architecture smells. Titan tool-set (Xiao et al. 2014) detects modularity violations such as cyclic dependencies. Similarly, Arcan (Fontana et al. 2017) detects two architecture (*cyclic dependency* and *unstable dependency*) and one design smells (*hub-like dependency*). Mo et al. (2015) identify a set of hotspot patterns (recurring architecture problems) based on a combination of historical and architectural information of software systems. Hochstein and Lindvall (2003) describe a method of diagnosing architectural degeneration using an architecture evaluation tool. Nam et al. (2018) present a method to visualize the evolution of software architecture that may lead a manual observer to spot architecture smells. ARCADE (Behnamghader et al. 2017) allows architecture recovery and computing of architecture decay and change metrics to observe and track architecture evolution. Despite the above-mentioned attempts, to the best of our knowledge, none of the existing publicly available tools can detect a decent number of architecture smells in large software systems.

There are a few commercial tools such as Lattix³, Sotoarc⁴, and Structure101⁵ that support architecture comprehension and conformance. Although some architecture smells (such as *cyclic dependency*) could be detected using these tools, they do not support architecture smell detection natively and explicitly. For design smells, tools such as NDepend⁶ and JArchitect⁷ help users identify quality issues. The analysis provided by these tools may aid the users in spotting some specific design smells (such as *unutilized abstraction*, *insufficient modularization*, and *cyclically-dependent modularization*). However, the tools lack explicit support for design smells. Table 1 provides a brief overview of the features supported by various commercial tools.

To summarize, the presented study provides a useful tool for practitioners and researchers to detect architecture and design smells, but also explores the relationships among them by analyzing a large number of repositories. The study presented in this paper is different from the existing research in a variety of ways. Firstly, we explore the relationships (through correlation, collocation, and causality analysis) between architecture- and design-level smells, whereas the previous studies focused on identifying predominantly the collocation of design smells. We believe that studying smell relationships horizontally (*i.e.*, smells belonging to

³<https://lattix.com/>

⁴<https://www.hello2morrow.com/products/sotograph/sotoarc>

⁵<https://structure101.com/>

⁶<https://www.ndepend.com/>

⁷<https://www.jarchitect.com>

Table 1 Comparison of commercial tools for design/architecture analysis

Tools	Design/Architecture smells natively supported	Design/Architecture smells indirectly supported	Other features
NDepend	Insufficient modularization, multifaceted abstraction, unutilized abstraction, deep hierarchy	Code quality metrics, dependency matrix, complexity analysis, code query	Quality gate, trend monitoring, continuous integration reporting
JArchitect	Insufficient modularization, multifaceted abstraction, unutilized abstraction, deep hierarchy	Code quality metrics, dependency matrix, code query	Quality gate, trend monitoring
Lattix Architect		Dependency matrix	Architecture compliance, continuous monitoring
SotoArc		Interactive dependency visualization	Architecture compliance, refactoring simulation
Structure101		Interactive dependency visualization, complexity analysis	Architecture compliance
Designite	19 design and 7 architecture smells	Code quality metrics, dependency matrix	Trend analysis

the same granularity) as well as vertically (*i.e.*, smells belonging to different granularities) will complement the existing literature and reveal new insights. Secondly, where a number of previous studies (Abbes et al. 2011; Palomba et al. 2018; Walter et al. 2018) infer that the collocation of code smells complicates source code comprehension and maintenance, the collocation of smells at different design granularities might serve as a blessing in disguise as an occurrence of design granularity smell may indicate the occurrence of an architecture granularity smell, providing useful pointers to the developers for design optimization through effective refactoring. Thirdly, this is a mining study involving a relatively large set of more than three thousand open-source C# repositories including 26 design and architecture smells, which is expected to make the findings more generalizable. To the best of our knowledge, none of the existing research attempts targets to explore the temporal causality as well as correlation and collocation relationships between individual architecture and design smells.

4 Theoretical Background

In this section, we provide definitions of the considered architecture and design smells. Our presentation of the considered seven architecture and 19 design smells is based on the existing, considerable, coverage by the research and practitioner communities. Further, we elaborate on the detection mechanism of architecture and design smells that we use in our implementation.

4.1 Smell granularities and terminology

Code smell is an umbrella term used to describe indicators of poor quality that may occur in source code. Smells are classified based on various criteria (Sharma and Spinellis 2018b) such as domain, granularity, and artifact. In the traditional source code context, software smells are classified based on their scope and impact in the following three granularities: implementation, design, and architecture. Implementation smells are confined to typically a method. For instance, *long method*, *long parameter list*, or *complex conditional* (Fowler 1999) are implementation smells that can be detected just by observing the source code of a method. The granularity of design smells is higher than implementation smells where we observe classes, their properties, and their relationships with other classes to detect them. Examples of design smells include *insufficient modularization*, *multifaceted abstraction*, and *deep hierarchy* (Suryanarayana et al. 2014). Architecture smells occur at the highest granularity. We see source code in the form of components and their interactions with other components. *Feature concentration*, *god component*, and *dense structure* are examples of architecture smells (Garcia et al. 2009). In this study, a component refers to a *namespace* or an *assembly* in C# (equivalent to a *package* construct in Java) and a *type* or an *abstraction* refers to a *class* or an *interface*. Though the above discussion applies to object-oriented context, it can be generalized to apply on non-object oriented source code.

Many smells belonging to different granularities have similar names or causes. Despite the similarity, these smells are different because their scope and impact vary significantly. For example, *cyclic-dependency modularization* and *cyclic dependency* arise at design and architecture granularities, respectively. Although their names are almost the same, their semantics differ. Cycles (either directly or indirectly) among classes lead to *cyclically-dependent modularization* smell at design granularity whereas a cycle formed by two or more components introduces *cyclic dependency* architecture smell. The difference between

the two lies in the granularity and the level of abstraction. Similarly, though *multifaceted abstraction* and *feature concentration* smells share the cause *i.e.*, poor cohesion, they arise at design and architecture granularities respectively. Despite the same cause, their scope and impact differ significantly—*multifaceted abstraction* is the result of poor cohesion in a class whereas *feature concentration* shows poor cohesion among a set of classes belonging to a component. Apart from the differences arising from scope and impact, their detection mechanism also differ. For instance, *multifaceted abstraction* gets detected by computing Lack of Cohesion among Methods (LCOM), which takes into account the commonalities (such as common field access) among the methods of a class. On the other hand, *feature concentration* architecture smell is detected by computing Lack of Component Cohesion (LCC) which considers similarity among classes (refer to Sections 4.2 and 4.3).

In this study, we restricted the scope of the analysis to source code; thus, we are analyzing source code to detect smells at different granularities. A set of architecture smells can also be detected by comparing design artifacts, such as architecture diagrams or design specifications, with the source code. However, it is not feasible to find architecture diagrams or specifications for open-source projects. Moreover, even if they are available, they significantly differ in notation as well as form and format that makes it practically infeasible to consider them for the purpose.

4.2 Architecture Smells

The architecture smells selected for this study represent anomalies related to a variety of internal software quality aspects. For instance, *feature concentration* smell represents *cohesion*, *scattered functionality* represents *coupling*, *god component* captures *size*, and *dense structure* represents *complexity* aspects. We implemented the support to detect architecture smells in Designite for this study. We also considered the automation feasibility of smell detection while choosing these smells and hence did not pick architecture smells that are infeasible or too complex to be automated. One example of such a smell is *stovepipe system* architecture smell (Brown et al. 1998) which is detected when subsystems of an application are integrated in a temporary manner using multiple integration strategies and mechanisms.

The architecture smells considered in this paper along with their detection mechanisms are explained below. The detection mechanism used in the tool is in-line with the existing literature. For instance, *unstable dependency* smell gets detected by computing instability metric of each component, which is proposed by Martin (2002) and used by Francesca et al. (2016, 2017). Some of the smells are implemented for the first time in the software engineering literature as far as we know; hence, for them we not only provide detection mechanisms but also discuss the rationale used to decide any threshold they use. Taking a deeper look at the thresholds reveals that there are two kinds of thresholds; apart from the key metric thresholds, sometimes additional conditions are imposed to reduce the possibilities of false-positives. For instance, ambiguous interface smell gets detected when a component has only one public method. However, this heuristic will lead to many small components to be detected as the components suffering from this smell. Hence, to avoid this, we apply an additional check to ensure we don't detect too small components as smelly components.

4.2.1 Cyclic Dependency

This smell arises when two or more architecture-level components depend on each other directly or indirectly (Mo et al. 2015; Lippert and Roock 2006).

Detection mechanism:

- We compute a dependency list for each component. Therefore, such a list for component *A* represents the components on which component *A* depends.
Component *A* depends on component *B* if at least one of the classes in *A* refer (by association, aggregation, or composition) to at least one of the classes in component *B*.
- We construct a directed graph using the above information where *nodes* refer to components and *edges* refer to their dependencies.
- We then apply depth-first search algorithm to detect cycles in the graph for each component. For large graphs, we stop the exploration after a threshold (currently set to 5 hops) to avoid extraneous computation.

4.2.2 Unstable Dependency

This smell arises when a component depends on other less stable components (Fontana et al. 2016). Stable Dependencies Principle (SDP) (Martin 2002) states that the dependencies between packages should be in the direction of the stability of packages. Hence, a package should only depend on packages that are more stable than itself. An *unstable dependency* architecture smell occurs when this principle is not followed.

Detection mechanism:

- Instability of a component is computed as follows:

$$I = \frac{C_e}{C_e + C_a} \quad (1)$$

Here, I represents the degree of instability of the component, C_a represents the afferent coupling (or incoming dependencies), and C_e represents the efferent coupling (or outgoing dependencies).

- We compare the computed metric value of each component against its dependent components, and detect the smell when the dependent component is more stable.

4.2.3 Ambiguous Interface

This smell arises when a component offers only a single, general entry-point into the component (Garcia et al. 2009). This smell typically appears in event-based publish-subscribe systems where interactions are not explicitly modeled and multiple components exchange event messages via a shared event bus.

Detection mechanism: We detect this smell when we find a component containing only one *public* or *internal* method. An *internal* method in C# has the visibility inside the assembly; hence, other components within the assembly may access it. We detect the smell only when the component has at least 5 classes to avoid small components from getting reported as ambiguous interfaces. This additional check is applied to avoid false-positives derived from reported false-positive cases from users of Designite.

4.2.4 God Component

This smell occurs when a component is excessively large either in terms of Lines Of Code (LOC) or the number of classes (Lippert and Roock 2006).

Detection mechanism: We detect this smell when a component has more than 30 classes or 27 000 LOC following the recommendations by Lippert and Roock (2006).

4.2.5 Feature Concentration

This smell occurs when a component realizes more than one architectural concerns or features (de Andrade et al. 2014). In other words, the component is not cohesive.

Detection mechanism:

- The software engineering literature has relied on metrics such as Lack of Cohesion of Methods (LCOM) (Chidamber and Kemerer 1994) to identify diverse features or responsibilities realized by a class. We have extended the same concept for detecting the feature concentration smell.

We compute Lack of Component Cohesion (LCC) to measure a component's cohesion.

- To compute LCC, we identify related classes in a component, prepare a dependency graph, and identify the number of disconnected sub-graphs. Two classes are related if they share any of the association, aggregation, composition, or inheritance relationships.

$$LCC = \frac{\text{Number of disconnected sub-graphs}}{\text{Total number of classes}} \quad (2)$$

- We detect this smell if LCC is more than a pre-defined threshold. We currently use 0.2 as the LCC threshold to detect this smell. We chose this threshold after experimenting with various threshold values and analyzing the resultant set of detected smells manually. Since this smell has never been detected in software engineering literature, we derive the metric threshold by the following heuristic. If all the classes are connected to other classes in a component, the resultant dependency graph would be a connected graph and we will get $LCC = 0$. Hence, any component having $LCC > 0$ can be considered as a candidate for this smell. However, it would be a very stringent threshold. We experimented with a few options (0, 0.1, 0.2, and 0.3) to find out a suitable threshold by comparing the results with these options and found 0.2 as the optimal threshold. However, we allow other users and researchers to change the threshold in the tool as per their needs.

4.2.6 Scattered Functionality

This smell arises when multiple components are responsible for realizing the same high-level concern (Garcia et al. 2009). It is an indication that possibly classes or methods must be moved from one component to another in order to reduce coupling among components and enhance cohesion within each component.

Detection mechanism:

- We determine the accesses to at least two external components that occur from a method.
- When two or more components are accessed from one method, it becomes a possible case of scattered functionality smell. However, to ensure that we are not reporting the one-off instances (and thus false-positives), we identify the smell when such accesses occur more than one.

4.2.7 Dense Structure

This smell arises when components have excessive and dense dependencies without any particular structure (Sharma et al. 2016a).

Detection mechanism:

- This smell occurs when components form a very dense dependency graph. In order to detect this smell, a dependency graph involving all the components is formed and the average degree of the graph is computed. The average degree of a graph can be computed as follows:

$$\text{Average degree} = \frac{2 \times |E|}{|V|} \quad (3)$$

Where E is the set of all edges and V is the set of all vertices belonging to the graph.

- We detect the smell when the average degree is greater than a pre-defined threshold. To choose an appropriate threshold, we started with the 7 ± 2 rule of psychology (Miller 1956). We experimented with values 5 to 9 for the threshold value and analyzed the resultant set of detected smells manually. We found the threshold value 5 as the most appropriate.
- Since the dependency graph is formed by considering all the components present in a repository, at most only one instance of this smell can occur for a source code repository.

4.3 Design Smells

Suryanarayana et al. (Suryanarayana et al. 2014) proposed a mechanism to classify design smells identified in the literature based on four major object-oriented principles (*i.e.*, *abstraction*, *encapsulation*, *modularization*, and *hierarchy*). We detect 19 smells, all that is feasible to implement without detecting too many false-positives, out of 25 smells proposed in the catalog. This set of supported smells covers all the commonly known and studied design smells. Though, the design smells detection support in Designite has been discussed in earlier work (Sharma et al. 2016b, 2017), we elaborate on the detection mechanism for all the considered design smells. As the case with architecture smells, we provide rationale for choosing certain threshold where the smells are not commonly detected and their corresponding thresholds are not known.

4.3.1 Duplicate Abstraction

This smell arises when two or more abstractions have identical names or identical implementations. Duplicate code is difficult to maintain. Often, a change in one of the clones needs to be reflected across all the other duplicates. Overlooking this implied overhead may lead to difficult to trace bugs. This smell can be refactored by identifying and extracting a common implementation in the duplicates into a common class or method (Suryanarayana et al. 2014).

Detection mechanism: We detect this smell when we find code clones (type-1) (Koschke 2007) of size greater than 20 lines. The size is chosen as 20 lines to avoid detecting smaller auto-generated code snippets as clones.

4.3.2 Imperative Abstraction

This smell arises when an operation is turned into a class. If operations are turned into classes, the design suffers from an explosion of one-method classes and complexity of the design increases. This smell can be refactored by creating an appropriate abstraction to host the method existing within the imperative abstraction (Suryanarayana et al. 2014).

Detection mechanism: If a class has only one public method and the size of the class (in terms of LOC) is greater than the pre-defined threshold (*i.e.*, 100), we detect this smell. We applied this threshold to avoid detecting smaller classes which contain only one method but do not justify the rationale of getting converted the class into a method.

4.3.3 Multifaceted Abstraction

This smell arises when an abstraction has more than one responsibility — contract or obligation (Booch et al. 1999, p. 53) — assigned to it. When an abstraction includes multiple responsibilities, it implies that the abstraction will be affected and needs to be changed for multiple reasons. The smell can be refactored by applying “extract class” refactoring on a subset of methods to form more cohesive classes (Suryanarayana et al. 2014).

We clarify now in the manuscript that term ‘responsibility’ (which is coming from object-oriented parlance) means the abstraction’s contract or obligation. It also reflects in Single responsibility principle. Software engineering literature is using the LCOM metric to measure the (lack of) cohesion that essentially captures the interactions among the source code elements of the class and identifies the degree of cohesiveness.

Detection mechanism: Software engineering literature is using the Lack of Cohesion among Methods (LCOM) metric to measure the (lack of) cohesion that essentially captures the interactions among the source code elements of the class and identifies the degree of cohesiveness. To detect the smell, we compute (LCOM) metric for each type. If the value of the metric is greater than a threshold (*i.e.*, 0.80) and the type is not very small —number of fields and methods are greater than or equal to a threshold (*i.e.*, 7), we detect the smell. The threshold on LCOM is inspired by existing work (Filó and Da Silva Bigonha 2015) and a popular code quality tool NDepend⁸.

4.3.4 Unnecessary Abstraction

Each abstraction has two essential elements—data (*i.e.*, fields) and behavior (*i.e.*, methods). If an abstraction does not have associated methods but only fields, then that abstraction is not needed. This smell occurs when an abstraction that is actually not needed (and thus could have been avoided) gets introduced in software design. Having needless abstractions in the design increases its complexity unnecessarily and affects the understandability of the overall design. This smell is eliminated by removing the unnecessary abstraction or by applying “inline class” refactoring to merge the class with another class (Suryanarayana et al. 2014).

⁸<https://www.ndepend.com/docs/code-metrics#LCOM>

Detection mechanism: If a type (except enumerations) has no methods and the number of fields and properties are less than a threshold (*i.e.*, 5), we detect this smell. We derive this threshold because the number of fields and properties less than the threshold are commonly declared as a part of enumeration types or can be easily placed in the classes where they are getting used.

4.3.5 Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable). Unused abstractions pollute the design space and increase cognitive load. In addition, such abstractions may lead to reliability issues by getting invoked accidentally. An unused abstraction can be refactored by eliminating it from the code (Suryanarayana et al. 2014).

Detection mechanism: A type is unutilized if the fan-in metric of the type is zero *i.e.*, there are no classes that depend on this type and the type has no supertype. In case, the type has supertype, the type suffers from this smell if fan-in for both the type and its supertype is zero.

4.3.6 Deficient Encapsulation

Public fields are accessible across the software system and hence they make software design complex and debugging difficult. This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required. Presence of this smell indicates that the internal implementation details are exposed for the abstraction leading to reduced understandability due to the complex interface. Also, the clients of the abstraction may depend directly upon the implementation details of the abstraction making it difficult to change and extend the abstraction. It can be refactored by applying “encapsulate field” refactoring (Suryanarayana et al. 2014).

Detection mechanism: If a type has at least one public field or global field (declared as *public static*), we detect this smell.

4.3.7 Unexploited Encapsulation

This smell arises when explicit type checks are performed (using chained if-else or switch statements that check for the type of the object) instead of exploiting the variation in types already encapsulated in the form of an inheritance hierarchy. A hierarchy facilitates variations in an encapsulation; however when type-based switches are performed, the client code needs to be changed whenever an existing abstraction is changed. Hence changeability and extensibility of the software system are impacted. This smell can be refactored by replacing the type-based switches to an inheritance hierarchy and polymorphic methods (Suryanarayana et al. 2014).

Detection mechanism: We retrieve the list of types that are being explicitly checked in a method. We find the number of checked types that belong to the same inheritance hierarchy. If the number is greater than one, we detect this smell. The threshold value is chosen as greater than one because it doesn't make sense to detect the smell when there is only one subtype.

4.3.8 Broken Modularization

This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions. When members that should be put together are scattered across different abstractions, understanding all the methods and their functionality is no longer easy. Furthermore, introducing changes and feature enhancements becomes difficult because modifications may need to be made across several abstractions. To eliminate the smell, it is required to put together the data members and corresponding methods into a single abstraction by using a series of “move method” or “move field” refactorings (Suryanarayana et al. 2014).

Detection mechanism: There are two variants of this smell (Suryanarayana et al. 2014)—classes that are used as a holder of data members without corresponding methods and methods in a class more interested in members of another class. We only detect the first variant of the smell.

If a type doesn’t have any methods and count of fields and properties is greater than a certain threshold (*i.e.*, 5), we detect this smell. The additional threshold (5) is applied to ensure that we don’t tag small classes as broken modularization.

4.3.9 Cyclically-dependent Modularization

This smell arises when two or more abstractions depend on each other directly or indirectly creating a tight coupling between the abstractions. When a set of abstractions are coupled together in a tangle, a change in one of these abstractions may lead to a ripple effect across all the coupled classes. Hence, it is difficult to understand as well as introduce new features or changes to the classes belonging to the tangle. This smell can be refactored by breaking the cycle by moving some of the fields or methods to another class (Suryanarayana et al. 2014).

Detection mechanism: We prepare a dependency graph of types from their incoming and outgoing dependencies. We use this dependency graph to detect direct or indirect cycles.

4.3.10 Hub-like Modularization

This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions. Any change to an abstraction, on which many other abstractions depend, is difficult to change because the change in the abstraction may lead to changes in the abstractions that depend on this abstraction. Due to this reason, a hub class can be affected by numerous other abstractions, and can, in turn, affect abstractions that depend on that hub class. This coupling pattern makes the design with hub classes prone to ripple effects impacting the changeability and extensibility of the design. The smell can be refactored either by splitting the hub class using “extract class” or in some cases by applying “chain of responsibility” pattern (Suryanarayana et al. 2014).

Detection mechanism: If both fan-out and fan-in metrics of a type are greater than a threshold (*i.e.*, 20), the type suffers from this smell. This threshold is selected based on the recommended threshold by Ferreira et al. (2012).

4.3.11 Insufficient Modularization

This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both. A large and complex implementation is difficult for developers to comprehend and change. A bloated interface, which is often the case in this smell, is difficult to use by the client abstractions. Often, multiple kinds of clients access the abstraction and hence a change in this abstraction leads to breaking many of the client classes. The smell can be refactored by segregating related members and extracting them into separate classes using “extract class” refactoring (Suryanarayana et al. 2014).

Detection mechanism: There are three forms of this smell; hence, we report this smell when even one of the below case is true.

- If the count of public methods in a type crosses a threshold (*i.e.*, 20), the type is suffering from *insufficient modularization*.
- If the total methods in a type are more than a threshold (*i.e.*, 30), we detect this smell. We extrapolated the suggestion of Lippert and Roock (2006) of number of classes in a component to the number of maximum desired methods in a class.
- We compute Weighted Methods per Class (WMC) for each type. We detect this smell if the value of the metric is more than a threshold (*i.e.*, 100). The threshold is chosen based on the recommendation by Herbold et al. (2011).

4.3.12 Broken Hierarchy

This smell arises when a supertype and its subtype conceptually do not share an “IS-A” relationship resulting in broken substitutability. When a supertype and its subtypes do not honor “IS-A” relationship, clients of the abstractions may attempt to assign objects of subtype to supertype references and may expose to unexpected behavior. For example, invoking a method that is not honored by a subtype object may lead to a runtime failure. This smell can be refactored by applying “replace inheritance with delegation” refactoring (Suryanarayana et al. 2014).

Detection mechanism: The relation “IS-A” means that a type is a specialized version of its supertype. It implies that a subtype implements one or more methods in a more specific way. Our detection mechanism uses this implication to form the following heuristic. For each class that has at least one super class with at least one public method, we check whether the class satisfies the condition of *broken hierarchy* smell. If the class doesn’t have any method overridden or “leniently” overridden from its super classes, we detect the smell. A method is leniently overridden when the method name matches (but not necessarily the parameter types) with any of the public methods in the super classes; it is akin to method overloading within an inheritance hierarchy.

4.3.13 Cyclic Hierarchy

This smell arises when a supertype in a hierarchy depends on any of its subtypes. In the presence of this smell, any modification in subtypes may affect supertypes that may, in turn, affect other subtypes of the hierarchy. Refactorings such as “move method” and “extract

class” could be utilized to break the cycle. Also, in some cases, it is feasible to merge both the subtype and supertype into one type to remove the smell (Suryanarayana et al. 2014).

Detection mechanism: If a type accesses any of its subtypes, then we detect this smell.

4.3.14 Deep Hierarchy

This smell arises when an inheritance hierarchy is excessively deep. With the increase in the depth of a hierarchy, it gets difficult to predict the behavior of the leaf classes since such classes inherit a large number of public and protected methods. Refactorings such as “collapse hierarchy” and “split hierarchy” could be used to remove the smell (Suryanarayana et al. 2014).

Detection mechanism: We measure the Depth of Inheritance Tree (DIT) metric for each type. If the metric value crosses a threshold (*i.e.*, 6), as recommended by Suryanarayana et al. (2014), we detect this smell.

4.3.15 Missing Hierarchy

This smell arises when a code fragment uses conditional logic (typically in conjunction with “tagged types”) to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations. It is easy to modify existing types and add support for new types within the hierarchy when a hierarchy is encapsulating variation. However, with type-based switches, the client code needs to undergo a change whenever an existing type is changed. This smell can be refactored by applying “extract hierarchy” refactoring (Suryanarayana et al. 2014).

Detection mechanism: We get a list of types checked explicitly in a method, for example, by using *instanceof* operator. Then, if more than one of the types in this list are not belonging to an inheritance hierarchy then we conclude that an inheritance hierarchy is missing.

4.3.16 Multipath Hierarchy

This smell arises when a subtype inherits both directly as well as indirectly from a supertype leading to unnecessary inheritance paths in the hierarchy. Redundantly inherited supertype complicates the hierarchy leading to confusion and in turn impacts readability and understandability quality attributes. The smell can be refactoring simply by removing the unnecessary inheritance paths in a hierarchy (Suryanarayana et al. 2014).

Detection mechanism: We derive a list of the direct super classes of a class. We also retrieve all the ancestors of all the parents. If there is any type common between these two lists, we conclude the presence of this smell.

4.3.17 Rebellious Hierarchy

This smell arises when a subtype rejects the methods provided by its supertype(s). Rejecting methods from supertypes indicates that the subtype cannot be safely replaced with a reference of supertype and hence violates the Liskov substitution principle (Martin 2002).

When the smell is present, the client of the abstraction may be surprised why the subtype objects do not behave as expected. Furthermore, with this smell, clients cannot freely substitute supertype references with subtype objects. This constraint leads to the tight coupling of client code with the concrete types of the hierarchy and impacts the changeability of the design (Suryanarayana et al. 2014).

Detection mechanism: We check all the non-private methods in a class. If any method is overridden and either the method is empty or has only statement that throws an exception, we detect this smell.

4.3.18 Unfactored Hierarchy

This smell arises when there is unnecessary duplication among types in a hierarchy. In the presence of this smell, a change in the clone code needs to be replicated across all the associated clones. This constraint not only impacts changeability of the system but also reliability (when the same change is not done in other associated clones). This smell can be refactored by “pull-up method” refactoring to remove the duplicates within a hierarchy (Suryanarayana et al. 2014).

Detection mechanism: We detect this smell when we find code clones in sibling types (where the classes share supertype).

4.3.19 Wide Hierarchy

This smell arises when an inheritance hierarchy is too wide indicating that intermediate types may be missing. In the absence of intermediate abstractions, the sibling classes are not the same level of abstraction. That makes it difficult for programmers to understand the sibling classes since some classes are too high-level while others are at a low-level of abstraction. Furthermore, the smell affects the extensibility of the hierarchy due to missing “hook points” facilitated by a properly designed hierarchy. A common strategy to remove the smell is to apply “extract superclass” refactoring to introduce intermediate abstractions (Suryanarayana et al. 2014).

Detection mechanism: We compute the Number of Children (NC) metric for each type. If the value of the metric crosses a threshold (*i.e.*, 10), as recommended by Suryanarayana et al. (2014), we detect this smell.

5 Study Design

We define five research questions exploring the relationship between architecture and design smells. In order to provide answers to the research questions, we specify a protocol to select and download open-source repositories. We analyze the downloaded repositories using Designite (Sharma et al. 2016b, 2018a)—our smell detection tool that we use in this study.

5.1 Research Questions

Building on the motivation and the background, we compose five research questions to explore the relationships between architecture and design smells. Figure 2 puts together the research questions pictorially.

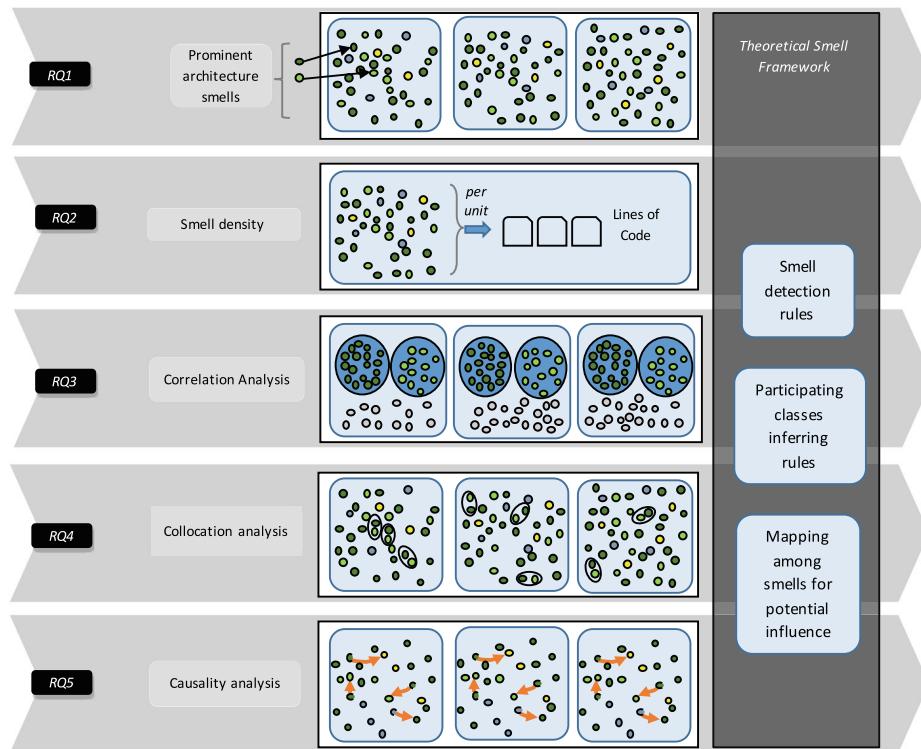


Fig. 2 Overview of the research questions

RQ1. Which architecture smells are more prominent in open-source C# repositories?

RQ1 aims to identify whether a specific sub-set of considered architecture smells is more prevalent in the analyzed open-source software repositories. The answer to RQ1 may caution the developers about a set of architecture smells expected to have more chances of occurrence and prompts them to take precautionary measures.

RQ2. Is architecture smell density affected by the size of the repositories?

It is commonly believed that the complexity of a software system increases with the size of the system. In this research question, we aim to analyze whether the density of architecture smells increases or decreases with the increase in the system size. Smell density (Sharma et al. 2016a, 2017) is a normalized metric that represents the average number of smells identified per thousand lines of code.

RQ3. Are detected architecture smell instances correlated with design smell instances?

RQ3 explores how the architecture smells correlate with design smells. First, we consider the cumulative values of both kinds of smells (architecture and design). A strong correlation between kinds of code smells would encourage us to understand the occurrence patterns and provide valuable insights into the similarity between these pairs.

Further, we also investigate the correlation between individual design smells and architecture smells. This would help us to find out whether there exist specific types of design smells that are strongly correlated to specific types of architecture smells.

RQ4. Are architecture smells collocated with design smells?

RQ4 aims to explore whether architecture and design smell occur at the same location (*i.e.*, classes) within the source code. A positive result of the collocation analysis would establish a strong relationship between architecture and design smells.

Apart from exploring collocation cumulatively between both kinds of smells, the question investigates the collocation relationships between individual pairs of design smells and architecture smells. This would help us to figure out whether and to what extent specific design smells show collocation with architecture smells.

RQ5. Do design smells cause architecture smells (or vice-versa)? This research question extends the relationship analysis beyond correlation and collocation and explores whether design smells “cause” architecture smells or vice-versa. A smell d causes smell a if predictions of the values of a based on its own past values and on the past values of d are better than the predictions of a calculated only using its own past values (Granger 1969).

It would reveal the degree of influence that design smells have on architecture smells (and vice versa). A high influence would hint that by carrying out a suitable refactoring early during the evolution of the software may lead to fewer smells at the different granularity.

To address the research questions defined above, we design an experimental setup, perform the experiments, extract the required data, and document our inferences and observations.

5.2 Downloading Subject Systems

We used the following protocol to identify our subject systems.

- We use RepoReapers (Munaiah et al. 2017) to filter out low-quality and too small repositories among the abundant repositories present on Github. RepoReapers analyzed a huge number of GitHub repositories and evaluated each of the repositories on eight dimensions providing a fair idea about their quality characteristics. These dimensions are architecture (as evidence of code organization), continuous integration and unit testing (as evidence of quality), community and documentation (as evidence of collaboration), history, issues (as evidence of sustained evolution), and license (as evidence of accountability). A repository scoring low on these dimensions are indeed of low quality compared to a repository scoring high on the same dimensions. RepoReapers assigns a score corresponding to each dimension.
- We select all the repositories containing C# code where at least six out of eight RepoReapers’ dimensions have suitable scores. We consider a score suitable if it has a value greater than zero. Values of some of the dimensions (architecture, documentation, issues, and unit test) range between 0 and 1, some other dimensions (license and continuous integration) may take either 0 or 1. Only dimensions history and community may take values minimum 0 to more than 1. Our goal was to filter out poor quality repositories and hence we selected the criteria where we consider values greater than zero as favorable values. Our choice of C# is motivated by the fact that a large part of the academic literature focuses on subject systems written in the Java programming language. It is desirable that we, as the research community, diversify our explorations and generalize our research. Our choice of C# is an attempt to fill the gap as empirical studies in programming languages other than Java.

Table 2 Characteristics of the analyzed repositories

Attributes	Values
Repositories	3 073
Components	114 706
Types	1 120 960
Methods	5 545 197
Lines of code (C# only)	118 699 236

- Next, we sort the repositories based on the number of assigned stars. We select repositories tagged with more than 10 stars.
- Following these criteria, we download and analyze more than 3 400 repositories using Designite (version 3.4.0). We could not analyze some of the repositories due to either missing external dependencies or custom build mechanisms (*i.e.*, missing standard C# project files). We successfully analyze 3 073 repositories.
- Software test code contains different types of smells (*i.e.*, test smells Deursen et al. 2001) which is not in the scope of this paper. Hence, we exclude the test code belonging to the selected software repositories from our empirical analysis.

A complete list of the selected C# repositories along with their analyzed results can be found online (Sharma 2019a). Table 2 presents some key characteristics of the selected subject systems.

5.3 Tool Support—Designite

Designite (Sharma et al. 2016b, 2018a) is a software design quality assessment tool for software systems written in C# programming language. The tool supports detection of 19 design and 11 implementation smells. For the purpose of this study, we implemented support to detect seven architecture smells within the tool. Apart from its GUI-based desktop application, Designite also offers a console application which is particularly useful for analyzing a large number of repositories automatically. Customization is one of the major features of the tool—a user can customize the way input source code is provided to the tool, certain smells to skip in an analysis session, or change thresholds that are used to detect certain smells. The tool offers free *Academic* licenses for all academic purposes.

The selection of smells to support is based on the commonality of these smells in the software engineering literature. Specifically, many authors have defined, discussed, and detected commonly occurring design smells; a mature consolidated catalog is proposed by Suryanarayana et al. (2014). Designite implements 19 out of the 25 design smells presented in the catalog. Similarly, we implemented architecture smells commonly discussed by the software engineering community (Bloch 2008; Fontana et al. 2016; Sharma et al. 2016a; de Andrade et al. 2014; Lippert and Roock 2006).

5.3.1 Manual validation

We conducted a manual validation to establish the accuracy of the tool. We chose a project DotNetOpenAuth.Core from a well-known open-source repository DotNetOpenAuth⁹ for the purpose. The selected project contains 16 663 LOC, 136 types,

⁹<https://github.com/DotNetOpenAuth/DotNetOpenAuth>

and 7 components. We sought help from two volunteers to carry out manual validation—one volunteer works in a software development company (three years of industry experience) and another volunteer is a Ph.D. student with one year of industry experience. Both the volunteers did not work of the analyzed repository in advance; however, they have hands-on experience on working with complex industrial solutions and have a fair idea of software architecture and code smells.

We enforced the following protocol for the validation.

- Each volunteer carried out the initial manual analysis individually without discussing it with another volunteer.
- Given their industry experience, they were aware of the basic concept of code smells and commonly known smells. Each volunteer picked all the considered design and architecture smells one by one and understood the semantics of the smell. We provided additional material to make their learning faster.
- Both the individuals went through all source code files one by one and checked the existence of each smell following the definition of each smell.
- While identifying smells, they were allowed to use IDE features such as *go to definition* and *list all references* as well as metrics generated from other tools.
- Once both the volunteers completed the analysis, the volunteers discussed their results, sorted out differences, and prepared a consolidated mutually agreed results. The consolidated results had 52 design and 18 architecture smells.
- At this point, they used Designite and analyzed the considered project and obtained a list of design and architecture smells.
- They compared the results obtained from the tool with their set of smells and tagged them as true-positive, false-positive, and false-negative. During this categorization, they observed that a subset of smells is identified by the tool which was not revealed by their manual analysis. They analyzed each of the smells in the subset and categorized them as well similar to the rest of the smells.

Table 3 presents the result of the manual validation showing smell instances detected by Designite and the consolidated set of smells identified by the volunteers. The table also shows number of false-positives and false-negatives that we found in this validation. The detailed report showing individual smells along with the names of each component or class where they occur along with corresponding classification can be found online (Sharma 2019b).

Interestingly, volunteers could not find all the legitimate smells manually; though, when the tool reported these instances, they classified them as true-positive. The highest number of smells that were missed by the volunteers are *cyclically-dependent modularization* and *cyclic dependency*. In this context, the volunteers found only unit-cycles *i.e.*, cycles involving only two classes or components. However, the tool reported cycles with different lengths. Volunteers verified all of these non-unit cycles and found them as true-positive. It implies that many smells go unnoticed even one actively looks for them; this observation emphasizes the importance of using tools.

The tool reported two false-positive instances of *unutilized abstraction*. Both of the instances are reported for exception types *i.e.*, classes that define custom exceptions. The tool could not resolve such instances of the types when they are thrown from a return statement.

The tool also failed to detect one instance of *broken modularization* smell. The volunteers classified the smell because the class only has a few data members and an empty

Table 3 Results of manual validation

Smells	Designite	Manual	TP	FP	FN
Broken Hierarchy	2	1	1	1	0
Broken Modularization	2	3	2	0	1
Cyclically-dependent Modularization	34	34	34	0	0
Duplicate Abstraction	9	9	9	0	0
Hub-like Modularization	1	1	1	0	0
Imperative Abstraction	9	9	9	0	0
Insufficient Modularization	5	5	5	0	0
Multipath Hierarchy	1	1	1	0	0
Rebellious Hierarchy	2	2	2	0	0
Unnecessary Abstraction	20	19	19	1	0
Unutilized Abstraction	5	3	3	2	0
Wide Hierarchy	4	4	4	0	0
Cyclic Dependency	13	13	13	0	0
Unstable Dependency	4	4	4	0	0
God Component	1	1	1	0	0
Feature Concentration	5	5	5	0	0
Scattered Functionality	3	3	3	0	0
Dense Structure	1	1	1	0	0
	121	118	117	4	1

constructor. However, due to the presence of the constructor, the tool did not identify the smell.

We compute precision and recall exhibited by the tool in the following way.

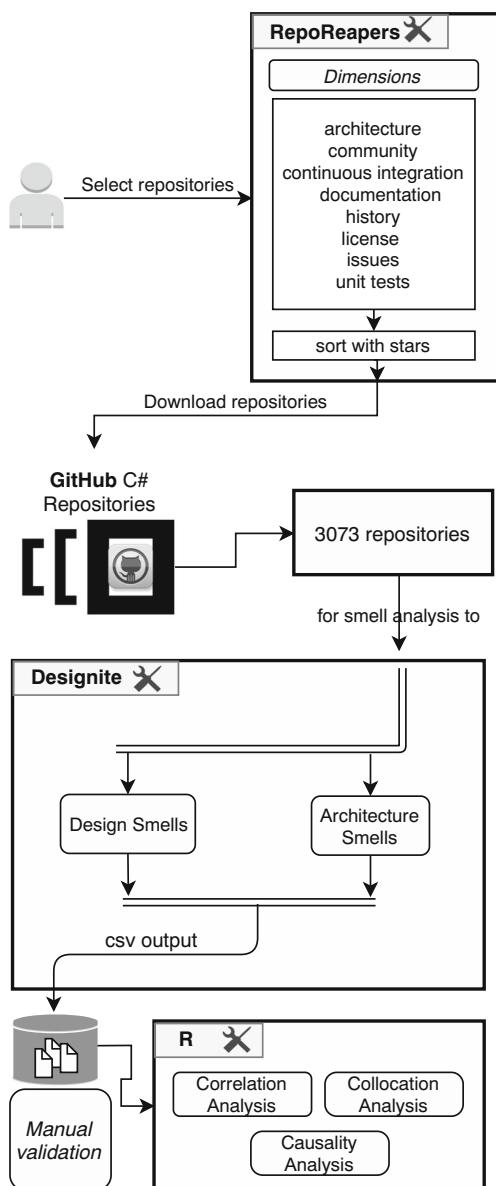
$$Precision = \frac{TP}{TP + FP} \quad (4)$$

$$Recall = \frac{TP}{TP + FN} \quad (5)$$

Here, TP, FP, and FN refer to true-positive, false-positive, and false-negative instances. Based on the above analysis, we obtain precision = $117/(117 + 4) = 96.6\%$ and recall = $117/(117 + 1) = 99.1\%$.

5.4 Overview of the Method

Figure 3 presents an overview of our experimental setup. The software repositories selected using RepoReapers are downloaded from GitHub. These repositories are then fed to Designite to identify smells that are exported in the form of csv files. We perform statistical analysis—specifically, correlation analysis (Spearman), Granger causality, and phi-coefficient using contingency matrix to answer the addressed research questions on the detected instances of architecture and design smells. We then infer, document, and present observations based on the results of our analysis.

Fig. 3 Overview of the Method

6 Results

This section presents the results along with our observations corresponding to each research question addressed in this study.

RQ1. Which architecture smells are more prominent in open-source C# repositories?

Approach: Designite offers a console application along with its GUI-based application to enable large-scale analysis. We use the console application and analyze each downloaded

Table 4 Number of detected instances and smell density (per KLOC) of architecture smells in the analyzed repositories

Architecture smell	#Instances	Smell density
Cyclic Dependency	47 270	0.398
Feature Concentration	23 483	0.198
Scattered Functionality	20 170	0.170
Unstable Dependency	11 823	0.099
God Component	6 758	0.057
Ambiguous Interface	1 223	0.010
Dense Structure	144	0.001

repository. The tool provides detected design and architecture smells in separate CSV files for each analyzed repository. We summarize the detected smells and compute the total number of detected smell instances for all the architecture smells.

Results: Table 4 lists the total number of architecture smells in all the analyzed repositories. The table reveals that the *cyclic dependency* is the most frequently occurring architecture smell followed by *feature concentration* smell. One potential reason for *cyclic dependency* to occur in a high volume is the permutations of the cycles due to one dependency that is mainly responsible for introducing a cycle. For example, Figure 4 shows dependencies among four components viz. Documentation (D), Semantics (S), TypeSystem (T), and TypeSystem. Implementation (I) of ICSharpCode.NRefactory project¹⁰. These four components have four unit cycles *i.e.*, cycles containing exactly two components; however, permutations of components forming a cycle having more than two components lead the tool to report eight *cyclic dependency* architecture smell instances in this project. The tool discards repeating sub-sequences and cycles more than the length of five.

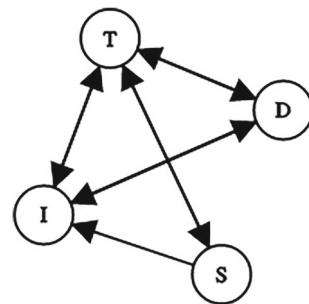
The *dense structure* smell has been detected the least number of times among the detected smells. This can mainly be attributed to the fact that, by definition, the smell can be detected at most once in a repository while all other architecture smells can be spotted multiple times in a repository. Along the same lines, the *ambiguous interface* smell occurs occasionally because often a component has multiple interfacing points and extra effort is required to make a component offer only one entry-point.

Apart from the number of detected instances of architecture smells, we present the total detected instances of design smells in the analyzed repositories in Table 5. At design granularity, *cyclically-dependent modularization* and *unutilized abstraction* are the most frequently occurring smells.

Both *feature concentration* and *multifaceted abstraction* capture the cohesion aspect. The difference between them is that *feature concentration* happens when a set of classes within an architectural component do not follow the single responsibility principle (SRP) whereas *multifaceted abstraction* smell arises due to the violation of the same principle among the methods of a class. Surprisingly, *feature concentration* smell occurs more often at architecture granularity (20% of the components) than its design granularity counterpart —*multifaceted abstraction* (0.3% of all the types). This indicates that components are more prone to violate the single responsibility principle than the classes at design granularity. Therefore, the software developers must pay attention to the component composition and cohesion when they extend a component.

¹⁰<https://github.com/icsharpcode/NRefactory>

Fig. 4 Dependencies among four components of NRefactory



Similarly, *god component* and *insufficient modularization* smells violate the principle of modularization at architecture and design granularity respectively. We observe that both of these smells show similar occurrence frequency; 6% and 5% of the components and types respectively suffer from *god component* and *insufficient modularization* smells.

In summary, *cyclic dependency* and *feature concentration* are the two most frequently occurring architecture smells. It is recommended for developers to avoid cycles among components and make components cohesive to increase maintainability of their software systems.

RQ2. Is architecture smell density affected by the size of the repositories?

Approach: Smell density (Sharma et al. 2016a, 2017) is a normalized metric that represents the average number of smells identified per thousand lines of code. We obtain a total number

Table 5 Number of detected instances and smell density (per KLOC) of design smells in the analyzed repositories

Design smell	#Instances	Smell density
Cyclically-dependent Modularization	224 218	1.888
Unutilized Abstraction	194 852	1.641
Duplicate Abstraction	165 558	1.394
Unnecessary Abstraction	157 619	1.327
Deficient Encapsulation	73 591	0.619
Insufficient Modularization	57 698	0.486
Broken Hierarchy	48 485	0.408
Broken Modularization	45 771	0.385
Unfactored Hierarchy	41 387	0.348
Rebellious Hierarchy	40 848	0.344
Imperative Abstraction	20 023	0.168
Cyclic Hierarchy	16 260	0.136
Unexploited Encapsulation	13 463	0.113
Wide Hierarchy	6 834	0.057
Multipath Hierarchy	6 269	0.052
Missing Hierarchy	3 296	0.027
Multifaceted Abstraction	3 166	0.026
Hub-like Modularization	1 509	0.012
Deep Hierarchy	630	0.005

of lines of code as well as total number of architecture smells for each repository from the results produced by the tool. Using this information, we compute architecture smell density for all the analyzed repositories individually. We also compute the Spearman correlation coefficient between the architecture smell density and the repository size (in LOC).

Results: We would like to observe how architecture smell density changes with the repository size. Figure 5a shows a scatter plot between architecture smell density and LOC of repositories. A Spearman correlation (ρ) value of 0.2140 (with p-value $< 2.2e - 16$) shows no correlation between architecture smell density and LOC. Similarly, analyzing the correlation between design smell density and LOC gives us a value of -0.0758 (with p-value $< 2.2e - 16$) (refer to Fig. 5b).

The results indicate that the size of a project has no impact on the smell density of the project.

RQ3. Are detected architecture smell instances correlated with design smells instances?

Approach: We compute the total number of smell instances belonging to each smell type individually—let us say A (total architecture smell instances) and D (total design smell instances) for each repository. We then find the Spearman correlation between the series of A and D . Further, we computed the total number of smell instances for each architecture (referred to as A_i where i may take values from 1 to 7 representing each architecture smell analyzed) and design (referred to as D_j where j may take values from 1 to 19 representing each design smell analyzed) smell. We calculate the Spearman coefficient between the individual pairs of architecture and design smells, *i.e.*, between each pair of A_i and D_j for all possible values of i and j , to observe the fine-grained correlation between architecture and design smell pairs.

Results: The number of detected instances of architecture and design smells exhibit a very high Spearman correlation coefficient (ρ) value of 0.8494 (with p-value $< 2.2e - 16$). Figure 6 shows a scatter plot between total detected instances of architecture and design smells in each repository. This indicates that architecture smells exhibit strong positive

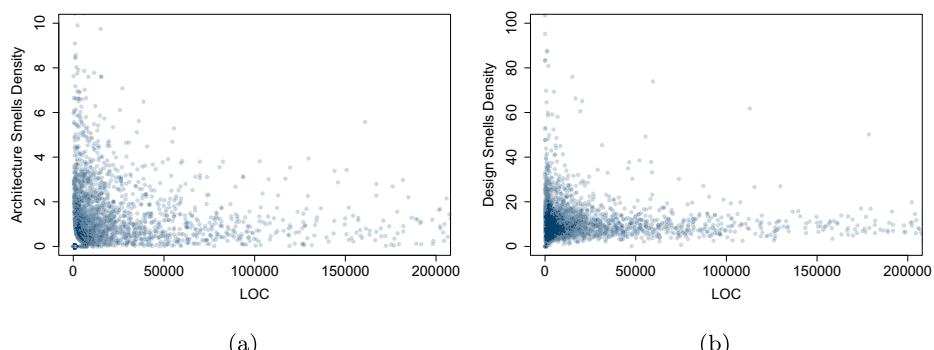


Fig. 5 Architecture/design smell density vs LOC

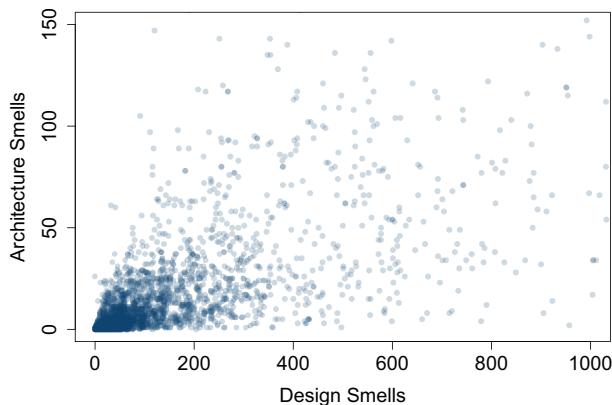


Fig. 6 Scatter plot between number of architecture smells and design smells

correlation with design smells. Therefore, it can be inferred that a large population of design smell instances present in a repository is associated with the presence of a high number of architecture smell instances and vice-versa. These observations might encourage a software developer to find and refactor architecture smells when she finds a large number of design smells in her software system.

To find deeper and fine-grained relationships, we compute Spearman correlation coefficients between individual design smells and architecture smells. Figure 7 presents Spearman correlation coefficient values for all 133 architecture-design smell pairs in a heatmap. The darker color of a cell in the heatmap shows a stronger correlation. A cell with coefficient value in the red color shows statistically insignificant values ($p\text{-value} \geq 0.005$).

The heatmap shows strong correlation for many smell pairs. *Feature concentration* and *unutilized abstraction* show the strongest individual correlation (0.83). We discussed a real-world example discussing the relationship between the two smells in Section 2. Smell pair *feature concentration* and *unnecessary abstraction* also shows a high correlation as the presence of *unnecessary abstraction* instances increases the chance of detecting *feature concentration* smell. Similarly, other pairs of smells that show high correlation are: *god component* and *insufficient modularization* and *cyclic dependency* and *cyclically-dependent modularization*. On the other hand, *ambiguous interface* and *dense structure* at architecture granularity and *deep hierarchy* at the design granularity show the least correlation with other smells.

The high correlation between cumulative as well as individual pairs of design and architecture smells indicates that a software developer must pay attention to the quality at architecture (or design) granularity if she perceives high number of smells at design (or architecture) granularity.

RQ4. Are architecture smells collocated with design smells?

Approach: Architecture and design smells differ in granularity, hence they get reported in a set of components and a set of classes, respectively. To analyze whether both kinds

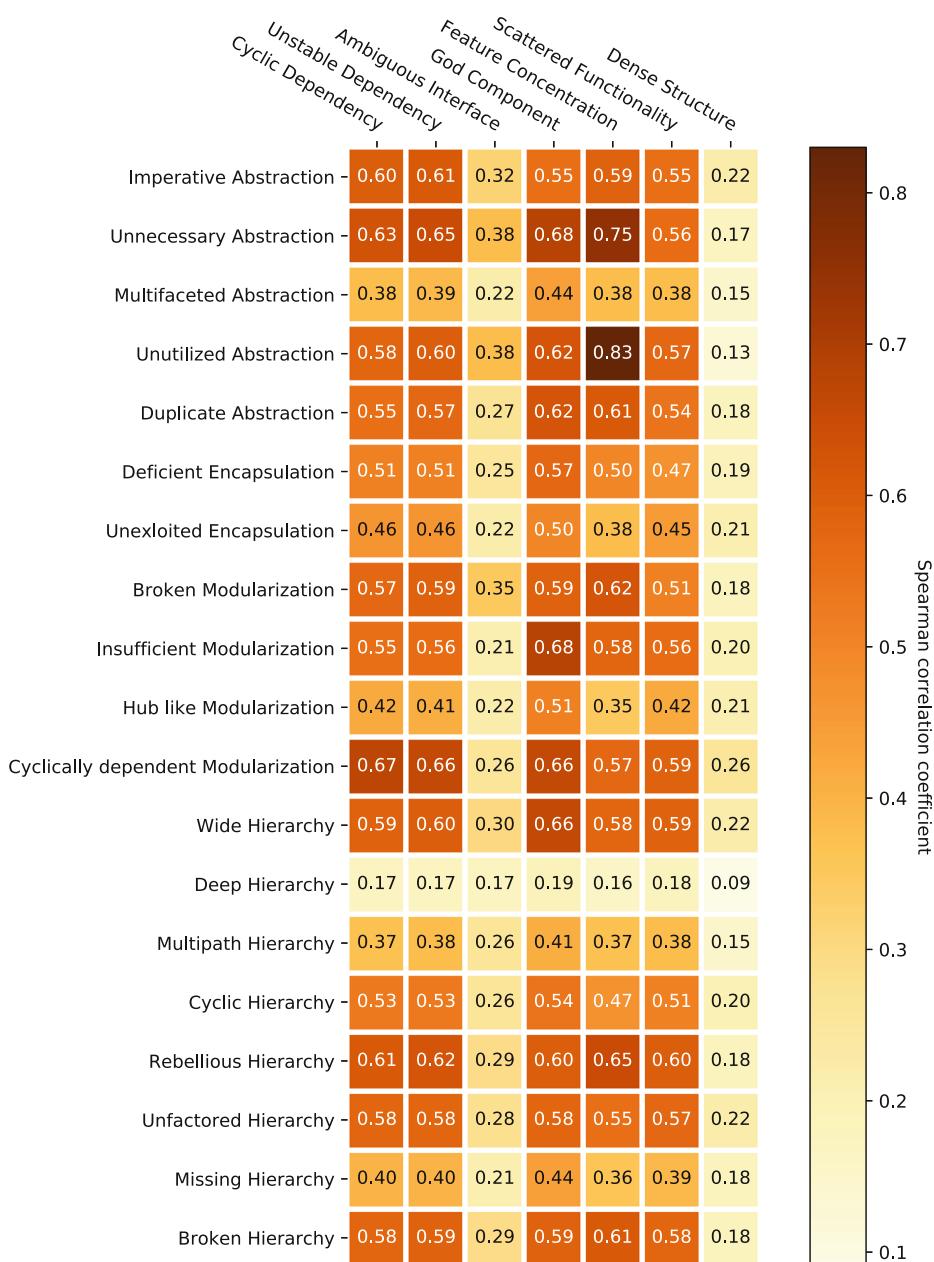


Fig. 7 Spearman correlation coefficients between individual architecture and design smells

of smells are collocated, we identify a set of *participating classes* for each architecture smell. A participating class contributes non-trivially to the occurrence of an architecture smell instance. Specifically, a design smell instance d and architecture smell instance a are considered to be “collocated” if a class reported by the instance d participates in instance a .

We created a table containing all the classes belonging to all the analyzed repositories with their corresponding total architecture and design smell instances. We created 2×2 contingency matrices for both the smell categories and compute phi-coefficient. The value of phi-coefficient measure the degree of association between two variables (Cramer 1946).

We analyzed cumulative instances of both smell categories as well as 133 individual smell pairs. Naturally, the frequency of architecture and design smells are not the same due to the difference in granularity and scale; thus the number of architecture smells is significantly lower than the number of design smells instances. We have to normalize both the numbers for semantically correct analysis and therefore we normalize the number of design smells by multiplying the ratio of specific design and architecture smells.

Table 6 shows the contingency matrix for a design and architecture smell pair. The values of variables a , b , c , and d are used to compute the phi-coefficient (Cramer 1946). However, as described above, we normalize the number of design smells instances *i.e.*, c .

$$c' = c \times \frac{\text{Number of architecture smells}}{\text{Number of design smells}} \quad (6)$$

We compute phi-coefficient using the following equation.

$$\phi = \frac{a \times d - c' \times b}{\sqrt{(a + b) \times (c' + d) \times (a + c') \times (b + d)}} \quad (7)$$

Inferring participating classes: To perform the collocation analysis between architecture and design smell instances, we identify participating classes for each architecture smell. A participating class contributes to the architecture smell non-trivially. We formulated and implemented the following rules to infer participating classes for each architecture smell.

Cyclic dependency: For each identified cycle, we find the classes (belonging to each component contributing to the formation of the cycle) that participate in the cycle. We include all these classes to the participating classes list.

Unstable dependency: This smell occurs when a component depends on another component which is less stable than itself. In this case, all the classes that refer to classes of a less stable component are the participating classes for this smell.

Ambiguous interface: We detect the smell when a component has only one public or internal method. We assign the class that has the public or internal method as the participating class for the architecture smell.

God component: The tool detects two variants of this smell. First, using LOC-based detection where LOC of the component crosses a threshold and second, using NOC-based detection where the number of classes in the component crosses a threshold. We include all the classes of the component as the participating classes.

Table 6 Contingency matrix for a design and architecture smell

		Design smell	
		1	0
Architecture smell	1	a	b
	0	c	d

Feature concentration: We detect the smell when a component is realizing more than one responsibility. We include all the classes of the component as the participating classes for the smell.

Scattered functionality: In this smell, classes scattered in multiple components realize the same architectural concern. We identify all these classes and tag them responsible for this architecture smell.

Dense structure: We identify all classes that refer to at least one class belonging to another component (hence contributing to the degree of the host component). We include all of the identified classes as the participating classes for the architecture smell.

Results: We obtained $\phi = -0.14$ as the value of phi-coefficient computed for cumulative values of design and architecture smells for each type. The value of phi-coefficient indicates that design and architecture smells do not collocate. We also compute phi-coefficients for individual architecture and design smell pairs. Figure 8 shows collocation analysis heatmap of architecture and design smells. Each cell shows the computed phi-coefficient for an architecture-design smell pair.

Phi-coefficient between *cyclically-dependent modularization* and *dense structure* shows the highest collocation. It is logical because cycles between classes increase the complexity of software's dependency graph and degree of components that in turn leads to *dense structure* smell. Similarly, the phi-coefficient between *feature concentration* and *unutilized abstraction* shows relatively high collocation. This collocation makes sense because the presence of one or more unutilized abstractions increases the value of LCC for a component. This increased value of LCC, in turn, leads to *feature concentration* smell as discussed in the detection mechanism for the smell (Section 4.2). We discussed a concrete example of collocation between *feature concentration* and *unutilized abstraction* in Section 2. Along the expected lines, *cyclically-dependent modularization* design smell shows relatively higher collocation with *cyclic dependency* architecture smell.

Apart from a few smell pairs, rest of the smells pairs show low values of collocation coefficient. The low collocation values suggest that majority of architecture and design smells do not collocate with each other.

We present an example of collocation of *feature concentration* architecture smell and *cyclically-dependent modularization* design smell. The component `ICSharpCode.NRefactory` in `NRefactory`¹¹ project contains ten classes/interfaces. As explained in Section 4.2, we identify disconnected dependency graphs within a component and compute LCC. Figure 9 shows the identified disconnected dependency graph where each box encapsulates a set of related classes. The computed LCC is 0.5 that makes the component suffering from *feature concentration* smell.

This component also reports a *cyclically-dependent modularization* design smell between `TextLocation` and `TextLocationConverter` classes (shown with red background).

RQ5. Do design smells cause architecture smells (or vice-versa)?

Approach: To further expand the analysis exploring the relationship between architecture and design smells, we explore the causality relationship between them.

¹¹<https://github.com/icsharpcode/NRefactory>

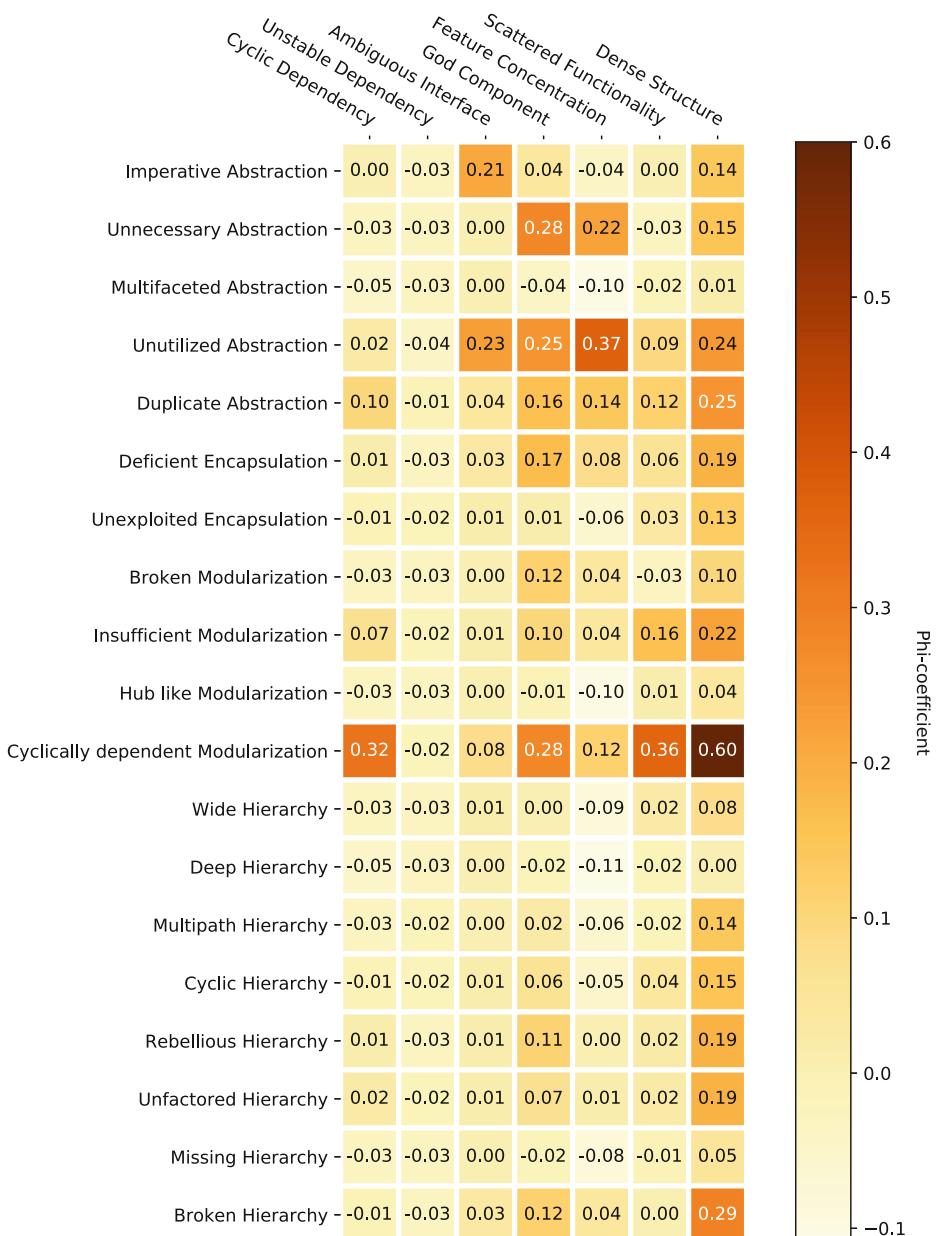
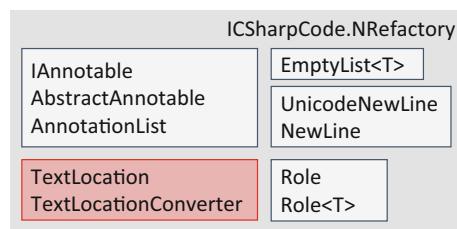


Fig. 8 Collocation analysis between individual architecture and design smell pairs

We use the mechanism proposed by Granger (Granger 1969) to figure out whether two random variables are associated with a causality relationship. We chose Granger's method to analyze causality relationship because the method has been used by other similar studies (Couto et al. 2013, 2014). Specifically, Palomba et al. (2018) have used the same method in their exploration to study co-occurrences among code smells.

Fig. 9 Feature concentration smell identified for the ICSharpCode.NRefactory component; the smell is collocated with *cyclically-dependent modularization* design smell (between TextLocation and TextLocationConverter classes)



Equation 8 presents Granger's method mathematically. Architecture and design smell instances computed over a period of time represent two time-series A and D respectively. A variable d , representing design smell instances at time t , causes variable a , representing architecture smell instances at time t , if the predictions of variable a with the past values of both a as well as d are better than the predictions using only the past values of a . The following equation expresses the above intuition mathematically.

$$a(t) = \sum_{j=1}^k f(a(t-j)) + \sum_{j=1}^k f(d(t-j)) \quad (8)$$

To carry out the causality analysis, we required time-series data of design and architecture smell instances detected for many versions belonging to a repository. We chose five repositories from our selected collection of repositories (refer to Section 5.2) with the highest number of commits. These repositories were chosen to maximize the chances of getting a sufficiently long time-series to carry out the analysis.

It is prohibitively resource-intensive to analyze all commits of a repository (one of the repositories contains close to 65 thousand commits). To select a subset of these commits, a naive way could have been to select a predefined number of commits equally divided on the temporal dimension. For instance, Couto et al. (2014) select commits at every two-weeks temporal distance. However, this mechanism would result in a set of commits that missed some commits with many changes and included commits that are superfluous *i.e.*, very similar to other commits. We propose a new mechanism to identify commits that need to be analyzed to produce a time-series for our causality analysis, based on a divide and conquer strategy.

Algorithm 1 presents the pseudo-code that we employed to select a set of commits for the analysis. It starts with the first and the last commit by including them in the selected commits. It then checks whether the commit in the middle of both the start and end commits should be included for the analysis. The algorithm includes the middle commit if the commit is significantly different than the start or end commit. A commit is significantly different than another commit if the number of classes that are different from one commit to another crosses a threshold (we used 20% as the threshold value). We decide a class in a commit is different than the class with the same name in another commit if both of these classes have different values for at least one of these metrics: weighted methods per class (WMC), number of children (NC), lack of cohesion in methods (LCOM), Fan-in, and Fan-out. If the algorithm finds a commit significant, then it looks for other commits that need to be included recursively by finding the next middle commit. We verified implementation of the algorithm by manually checking the produced results at each intermediary step to ensure the correctness of the algorithm.

Algorithm 1 The proposed algorithm to select commits for analysis.

Input: Path of a repository
Output: Selected commits

```

begin
    allCommits = GetAllCommitHash(repositoryPath)
    selectedCommit.Add(0)
    AnalyzeCommit(allCommits[0])
    lastCommitIndex = allCommits.Size – 1
    selectedCommit.Add(lastCommitIndex)
    AnalyzeCommit(allCommits[lastCommitIndex])
    FindIntermediateCommits(allCommits, 0, lastCommitIndex, ‘f’)
end
FindIntermediateCommits()
Input: all commits, startIndex, endIndex, direction
Output: Selected commits
begin
    if endIndex ≤ startIndex then
        | return
    end
    nextCandidate = (startIndex + endIndex) / 2
    if nextCandidate ≤ startIndex or nextCandidate ≥ endIndex then
        | return
    end
    AnalyzeCommit(allCommits[nextCandidate])
    if direction = ‘f’ then
        | isSignificant = IsDifferenceSignificant(allCommits[startIndex],
            allCommits[nextCandidate])
    else
        | isSignificant = IsDifferenceSignificant(allCommits[nextCandidate],
            allCommits[endIndex])
    end
    if isSignificant then
        | selectedCommits.Add(nextCandidate)
        | FindIntermediateCommits(allCommits, startIndex, nextCandidate, ‘f’)
        | FindIntermediateCommits(allCommits, nextCandidate, endIndex, ‘b’)
    end
end
IsDifferenceSignificant()
Input: commit1, commit2, threshold
Output: True/False
begin
    changedClasses = 0
    for each class c in commit1 and commit2 do
        metrics1, metrics2 = ReadMetrics(GetClass(c, commit1), GetClass(c, commit2))
        if metrics1 ≠ metrics2 then
            | changedClasses += 1
        end
    end
    changeRatio = changedClasses/totalClasses
    if changeRatio ≥ threshold then
        | return True
    else
        | return False
    end
end

```

Table 7 Causality test results (p-values) between cumulative values of design and architecture smells

	Architecture smells	Design smells
Architecture smells	1	0.0001
Design smells	0.2669	1

We analyzed each of the five repositories and present here the results from one repository, RavenDB,¹² with the largest number of significant commits.

It is mandatory to ensure the *stationary* property of a time-series before analyzing it and drawing conclusions based on that. A time-series is stationary if its statistical properties such as mean, variance, and autocorrelation are constant over time (Cox and Miller 1965). If a time-series is non-stationary, it shows seasonal effects, trends, and fluctuating statistical properties changing over time. Such effects are undesired for the causality analysis and thus a time-series must be made stationary before we perform the causality analysis. We carried out the augmented Dickey-Fuller unit root test (Fuller 1976) to check the stationary property of our time-series. Initially, our time-series was non-stationary. There are a few techniques to make a non-stationary time-series a stationary one (Kwiatkowski et al. 1992). We addressed this issue by applying a *difference* transformation, *i.e.*, subtracting the previous observation from the present observation for all columns. Techniques such as *differencing*, that we applied, help stabilize the mean of a time series by removing changes in the time series, and therefore eliminate or reduce the non-stationary nature of the series. This transformation resulted in a stationary time-series that we confirmed by performing the augmented Dickey-Fuller unit root test again. We carried out the causality analysis on the cumulative sum of all design and architecture smells for each analyzed commit as well as on individual types of design and architecture smells.

Results: We first compute the Granger causality between cumulative values of design and architecture smells. Table 7 presents the results of the causality test. The p-values shown in each cell show the causality from column to row. The table shows that design smells cause architecture smells with significant p-value (*i.e.*, less than 0.005). Time-series obtained from analyzing other repositories also confirm the observation. For instance, causality analysis on Umbraco-CMS¹³ repository also shows that design smells cause architecture smells with p-value 0.0036 while cumulative architecture smells do not decisively cause design smells with p-value 0.3593.

We also carried out causality analysis at individual smell granularity. Table 8 shows p-values corresponding to the causal effect of individual design smells on architecture smells, where the blue-colored cells indicate a significant causal relationship. It is interesting to note that *cyclically-dependent modularization* appears to cause all architecture smells. Also, smells such as *hub-like modularization* and *multipath hierarchy* also contribute towards causing three kinds of architecture smells each. On the other hand, many smells such as *deficient encapsulation*, *multifaceted abstraction*, and *unutilized abstraction* do not cause any architecture smell. *Deep hierarchy* smell was not detected in the repository and hence its degree of involvement in causing architecture smells cannot be determined.

We also performed causality analysis in the opposite direction *i.e.*, exploring whether architecture smells cause design smells at the individual level. Table 9 shows the results of

¹²<https://github.com/ravendb/ravendb>

¹³<https://github.com/umbraco/Umbraco-CMS>

Table 8 Causality test results (p-values) showing individual design smells causing architecture smells

	Imprecise interface	Uncoupled abstraction	Multiif action	Unin- itiated action	Dupli- cate abstrac- tion	Defi- cient encapsu- lation	Une- xploited action	Bro- ken encapsu- lation	Insufi- cient sula- tion	Hub- like mod- ula	Cycli- cally mod- ula	Wide- hierar- chy	Deep- path mod- ula	Multi- path mod- ula	Cyclic- licous hierar- chy	Re- lational hierar- chy	Uni- fied hierar- chy	Missing- ing hierar- chy	Broken hierar- chy
Cyclic Dependency	0.001	0.630	0.143	0.086	0.078	0.059	0.000	0.053	0.313	0.003	0.000	0.709	—	0.000	0.752	0.108	0.222	0.064	0.893
Unstable Dependency	0.003	0.376	0.403	0.128	0.069	0.069	0.000	0.048	0.827	0.003	0.000	0.726	—	0.000	0.205	0.264	0.121	0.123	0.714
Ambiguous Interface	0.004	0.019	0.181	0.323	0.044	0.144	0.353	0.026	0.027	0.011	0.001	0.001	—	0.943	0.192	0.285	0.509	0.384	0.231
God Component	0.265	0.003	0.064	0.337	0.001	0.757	0.583	0.007	0.046	0.070	0.000	0.741	—	0.765	0.050	0.504	0.114	0.001	0.305
Feature Concentration	0.323	0.013	0.128	0.611	0.001	0.318	0.329	0.004	0.407	0.226	0.000	0.841	—	0.650	0.027	0.325	0.174	0.005	0.222
Scattered Functionality	0.608	0.200	0.416	0.672	0.182	0.010	0.002	0.234	0.083	0.003	0.000	0.151	—	0.588	0.393	0.258	0.922	0.017	0.171
Dense Structure	0.253	0.105	0.404	0.058	0.217	0.275	0.138	0.416	0.644	0.208	0.001	0.000	—	0.095	0.061	0.180	0.002	0.004	—

Table 9 Causality test results (p-values) showing individual architecture smells causing design smells

	Cyclic Dependency	Unstable Dependency	Ambiguous Interface	God Component	Feature Concentration	Scattered Functionality	Dense Structure
Imperative Abstraction	0.724	0.649	0.737	0.882	0.479	0.875	0.000
Unnecessary Abstraction	0.000	0.000	0.549	0.834	0.952	0.255	0.041
Multifaceted Abstraction	0.007	0.144	0.387	0.121	0.237	0.086	0.025
Utilitized Abstraction	0.000	0.000	0.273	0.844	0.732	0.207	0.641
Duplicate Abstraction	0.024	0.000	0.543	0.889	0.826	0.314	0.722
Deficient Encapsulation	0.001	0.115	0.164	0.317	0.379	0.200	0.702
Unexploited Encapsulation	0.001	0.006	0.058	0.728	0.701	0.631	0.002
Broken Modularization	0.110	0.006	0.285	0.889	0.701	0.197	0.297
Insufficient Modularization	0.001	0.050	0.244	0.740	0.926	0.336	0.005
Hub-like Modularization	0.000	0.061	0.092	0.429	0.584	0.841	0.792
Cyclically-dependent Mod.	0.700	0.085	0.001	0.334	0.233	0.171	0.000
Wide Hierarchy	0.185	0.203	0.217	0.644	0.446	0.284	0.435
Deep Hierarchy	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Multipath Hierarchy	0.002	0.001	0.177	0.600	0.335	0.362	1.000
Cyclic Hierarchy	0.895	0.090	0.004	0.728	0.123	0.053	0.305
Rebellious Hierarchy	0.000	0.002	0.385	0.865	0.824	0.231	0.214
Unfactored Hierarchy	0.450	0.492	0.003	0.430	0.562	0.013	0.966
Missing Hierarchy	0.523	0.245	0.038	0.837	0.175	0.030	0.001
Broken Hierarchy	0.873	0.171	0.000	0.898	0.852	0.009	0.966

the analysis. Unsurprisingly, the *cyclic dependency* causes eight kinds of design smells. *God component*, *feature concentration*, and *scattered functionality* architecture smells do not contribute at all towards causing design smells. It is interesting to note that only *multifaceted abstraction* smell neither causes any architecture smell nor is it caused by any architecture smell.

In summary, we found that design smells cumulatively cause architecture smells. This implies that the current set of architecture smells in software systems are present due to smells at both design and architecture granularity in the previous versions of the software.

Next, we present couple of examples demonstrating design smells causing architecture smells as a software system evolves. Both the examples presented below are observed from the RavenDB repository; we analyzed many commits of the repository for the causality analysis.

As Table 8 shows, *cyclically-dependent modularization* exhibits the highest causality on architecture smells among all analyzed design smells. The following example shows how an instance of *cyclically-dependent modularization* that arose in one version leads to *cyclic dependency* architecture smell in a subsequent version of the software system. Figure 10 shows a design fragment where rounded-corner rectangles represent components and rectangles inside them represent classes belonging to the components. Orange arrows show dependencies between classes and black arrows show dependencies between components. Orange arrows

We observe that in a commit (ending with commit-hash bfacefc), a *cyclically-dependent modularization* design smell has been detected among five classes. In this commit, components *Serialization*, *Utilities*, and *Linq* are not forming a cycle and hence no *cyclic dependency* architecture smell involving these components is reported. However, in the next commit (ending with commit-hash 8cd11cff), a class *JPath* in *Linq* component refers to a class in *Serialization* component and hence introduces a dependency between the components (shown by a red arrow in Fig. 10b). It is important to note that source code changes carried out specifically in this commit do not introduce a cycle among architecture components. However, due to an existing cycle among other classes and components that was created in the last commit, a new cycle among the three components (*i.e.*, *Serialization*, *Utilities*, and *Linq*) is formed and reported in this commit. In other words, *cyclically-dependent modularization* design smell detected in the previous version **caused** a *cyclic dependency* architecture smell in this version.

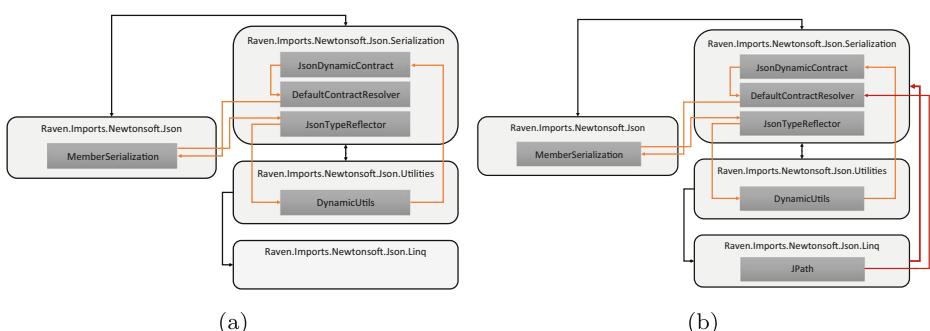


Fig. 10 cyclically-dependent modularization smell in version k leading to cyclic dependency architecture smell in version k+1. Dependencies between classes are shown by using orange arrows; black arrows show dependencies between components. Red arrow shows a new dependency that completes a cycle among three components

Similarly, we discuss below how *unnecessary abstraction* design smell detected in a version caused *god component* architecture smell in the following versions of the software system. In a commit (ending with commit-hash 77ee97d8), component `Newtonsoft.Json.Utilities` gets detected with four instances of *unnecessary abstraction* design smells; the component contains a total of 29 classes in the commit. However, in the following commit (ending with commit-hash ece305f5), the component introduces a few classes and the total number of classes reaches 32. Due to this increase, a *god component* architecture smell is detected in the component. However, we can infer that this instance of *god component* would not arise if the *unnecessary abstraction* smell instances were not detected in the previous commit. In summary, *unnecessary abstraction* smell instances detected in the previous commit **caused** the *god component* architecture smell instance in this commit.

7 Discussion and Implications

In this section, we provide interpretations of our findings. Our results reveal the individuality of smells at the explored granularities. In our context, individuality refers to the uniqueness of the smells showing differentiating characteristics than other smells. We also discuss the scope of each of the relationships examined in this work and the interplay among themselves. In the implication section, we emphasize that smells often occur independently at each granularity and software development teams must address them at all granularities.

7.1 Interpretations of results

We explore correlations between architecture and design smells cumulatively as well as between individual pairs. Very high correlations may indicate that a given smell is *superfluous*. For example, tracking humans' left-eye and right-eye colors will show an extremely high correlation between the two, and consequently storing only eye color is enough. Our analysis shows a very strong correlation between the high and low granularity smells when considered cumulatively; however, the fine-grained correlation analysis reveals varying degrees of correlation. This result demonstrates that each smell provides value-adding information. Furthermore, interestingly, even similar (by definition) smells at different granularities show varying degrees of correlation. Five architecture smells have similar corresponding smells at design granularity; it means that these smells represent and capture the same concept at different granularities. These smell pairs with their corresponding correlations are *cyclic dependency—cyclically-dependent modularization* ($\rho = 0.67$), *feature concentration—multifaceted abstraction* ($\rho = 0.38$), *scattered functionality—broken modularization* ($\rho = 0.51$), *god component—insufficient modularization* ($\rho = 0.68$), and *ambiguous interface—imperative abstraction* ($\rho = 0.32$). The varying degree of correlations—from low to moderate and strong—indicates the non-monotonic relationship between these smell pairs and further emphasizes the individuality and uniqueness of these smells.

We examine the collocation relationship between architecture and design smells; the results show that they exhibit collocation only for some smell pairs. It implies that though architecture smells arise from code and implementation choices made during the software development, the factors that cause them go beyond these choices and they have their individuality unique from smells at design granularity.

Our temporal analysis to explore the causality between design and architecture smells clearly shows that design smells cause architecture smells. In other words, the cause of the present set of detected architecture smells can often be traced back not only to the architecture smells in previous versions but also to the design smells.

7.2 Understanding the interplay between correlation, collocation, and causation

Let us recap the correlation, collocation as well as causation between design and architecture smells (see Fig. 11).

Correlation analysis is performed at the unit-scope of a repository *i.e.*, we measure the total number of detected instances for design and architecture smells for each repository and perform correlation analysis between the obtained series. This implies that the high correlation that our results exhibit is applicable at the repository or project level.

Collocation is calculated per class for each pair of design and architecture smell instance.

We determine if the considered pair of architecture and design smell instance is occurring in the same set of classes. Architecture smells are by default reported at the component granularity and hence we map each instance of architecture smell to a set of classes by computing participating classes for the smell.

Causation is a temporal relationship that we measure by analyzing many commits of a repository and performing Granger's causality on the obtained time-series.

All this means is that correlation and collocation analysis differ significantly on their unit-scope. Given a confined unit-scope *i.e.*, type, collocation represents a stricter relationship compared to correlation. The source of the weaker collocation can be traced back to the stricter relationship.

7.3 Implications of Reported Findings

We infer the following implications for the software development community.

Software development teams must detect, analyze, and refactor smells at all granularities. This implication is derived from our correlation analysis for smells arising at different

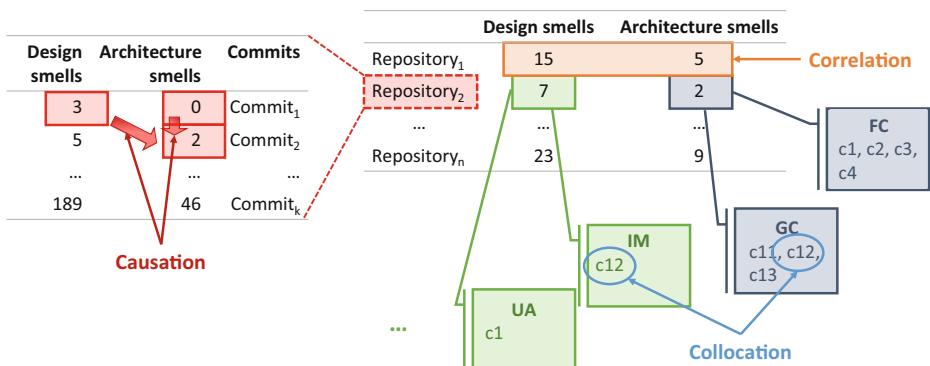


Fig. 11 Understanding correlation, collocation, and causation. IM and UA are examples of design smell instances and GC and FC are examples of architecture smell instances. c_i refers to involved class in a smell instance

granularities. Our results show that the presence of a high volume of design smells is associated with the presence of a high number of architecture smells and vice versa. Existing tools (such as NDepend¹⁴ and SonarQube¹⁵) mainly detect implementation and some design issues. Due to this limitation, a software development team using these tools perceives only a limited set of quality issues and thus issues at higher granularities may go unnoticed.

Developers must avoid cycles among classes as well as among components to keep the structure of the software easy to understand. Our results show that cyclic dependencies at both design and architecture granularities occur most frequently in open-source C# repositories compared to other smells (refer to RQ1 results). A higher number of cycles in software introduce tangles and make the software difficult to comprehend.

The smell density of a software system does not depend on the size of the software. Actively used software systems grow; however, whether the software evolves with the focus on code quality or not defines the long-term maintainability of the software. For example, in our analysis, the *dense structure* smell has been detected in fewer than 5% of the analyzed repositories. We observed that the median of LOC computed for all the analyzed repositories is 7 394 while it is 42 405 for the repositories where the smell has been detected. This indicates that the smell is less prone to occur in small repositories. However, the large size of a repository is not the only deciding factor. We found that 72 repositories are larger than the median LOC 42 405 where the smell does not occur. This implies that the evolution of a software system focused on quality may result in a more maintainable software system.

7.4 Secondary Contributions of this Work

We have added support to detect seven architecture smells in the Designite tool. The software development community may use this tool to analyze their source code and improve the maintainability of their code. The research community may utilize the tool to carry out studies concerning code smells. The tool is available online¹⁶ and free for all academic purposes.

We analyzed more than 3000 repositories to prepare a dataset containing the detected smells at architecture, design, and implementation granularities for each of the analyzed repositories. We used this dataset to answer the research questions addressed in this paper. We have made the dataset available online (Sharma 2019a). The software engineering research community may utilize it in many ways including bench-marking and comparison as well as exploring other dimensions of source code suffering from smells.

8 Threats to Validity

Construct validity concerns the appropriateness of observations and inferences made on the basis of measurements taken during the study. Static code analysis is typically prone to false-positives and false-negatives. To mitigate this concern, we employed a comprehensive set of tests for the smell detection tool used in this study to rule out obvious deficiencies. Additionally, we found the results of manual validation of the detected instances by the tool very satisfactory (Section 5.3.1).

¹⁴<http://www.ndepend.com/>

¹⁵<https://www.sonarqube.org/>

¹⁶<https://www.designite-tools.com>

Designite uses various metric thresholds to detect smells. It is a known and accepted fact that there is no one globally accepted threshold set for various metrics (Kessentini et al. 2014; Fourati et al. 2011). We chose the thresholds that are commonly used by the software engineering community. Moreover, we made many of the thresholds customizable within the tool to let users choose a set of appropriate thresholds based on their organization's preferences.

The higher the abstraction, the more important the context of a software system becomes. Context and domain knowledge play an important role while detecting and refactoring, especially, design and architecture smells. Given the sheer scale, it was not possible to carry out a qualitative analysis for all the repositories. Considering a large number of repositories mined in this study, we believe that the results are still relevant and generalizable.

While computing causality relationship between design and architecture smell instances over a period of time, there could be some confounding factors (VanderWeele and Shpitser 2013) that influence both design and architecture smell instances time-series. Randomization is often used to reduce the effects of confounding factors that we also adopted in the study.

External validity concerns the generalizability and repeatability of the produced results. The study analyzes only open-source C# repositories as subject systems. Given the fact that most of the current literature focuses only on subject systems written in Java programming language, our study complements the existing literature. Furthermore, we have considered a large set of 3 073 C# repositories of varied sizes and contexts, making this work the largest mining study (by scale) so far for software smells.

9 Conclusions

Architecture smells are the design degradation indicators at architecture granularity spanning multiple software components. Combining finer-grained code smells with the coarse-grain smells could make the task of maintaining a high quality of a software product easier. This work carries out correlation, collocation, and causation analysis to identify relationships among design and architecture smells. We implemented seven architecture smells in our code smell detection tool Designite and mined seven architecture and 19 design smells from a large set of 3 073 C# repositories downloaded from GitHub. A total of 1 232 348 architecture and design smell instances are made available to the software engineering research community in the form of a smell dataset.

The results of this study indicate that *cyclic dependency* is the most frequently occurring architecture smell. This may prompt developers to pay additional attention to avoid cycles among the components. The co-occurrence analysis shows that the architecture smells exhibit a strong positive correlation ($\rho = 0.85$) with design smells. This implies that a project containing a high number of design smells also exhibit a higher number of architecture smells and vice-versa. We perform fine-grained correlation analysis between individual smell pairs. The results reveal the varying degree of correlation between the smell-pairs belonging to different granularities.

Our collocation analysis reveals that both kinds of smells show selective collocation and the majority of smell pairs do not collocate with each other. Furthermore, our exploration to understand the causal relationship between architecture and design smells shows that design smells cause architecture smells. It implies that refactoring design smells early in a software development lifecycle may result in fewer architecture smells in the future versions of the software system.

In the future, we intend to extend our supported set of architecture smells and carry out more detailed analysis such as patterns of smell introduction and removal with time along with their relationships. Further, a comprehensive catalog of architecture smells containing examples, causes, appropriate thresholds, and remedies would be of great use to the software practitioners.

Acknowledgments The authors would like to thank Panos Louridas and Vasiliki Efstathiou from the Athens University of Economics and Business for providing useful improvement suggestions. We also like to thank Rohit Tiwari and Amit Dogra for putting numerous hours validating Designite.

This work was partially funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID). Grant agreement number 642954.

References

- Abbes M, Khomh F, Gueheneuc Y, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 2011 15Th european conference on software maintenance and reengineering, pp 181–190. <https://doi.org/10.1109/CSMR.2011.24>
- Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng* 21(3):1143–1191
- Bass L, Clements P, Kazman R (2012) Software Architecture in Practice, 3rd edn, Addison-Wesley Professional
- Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. *J Syst Softw* 107:1–14
- Behnamghader P, Le DM, Garcia J, Link D, Shahbazian A, Medvidovic N (2017) A large-scale study of architectural evolution in open-source software systems. *Empir Softw Eng* 22(3):1146–1193. <https://doi.org/10.1007/s10664-016-9466-0>
- Besker T, Martini A, Bosch J (2018) Managing architectural technical debt: A unified model and systematic literature review. *J Syst Softw* 135:1–16. <https://doi.org/10.1016/j.jss.2017.09.025>. <http://www.sciencedirect.com/science/article/pii/S0164121217302121>
- Bloch J (2008) Effective Java (2nd Edition) (The Java Series), 2 edn. Prentice Hall PTR
- Booch G, Rumbaugh J, Jacobson I (1999) The unified modeling language user guide. Addison-wesley, Reading
- Brown WJ, Malveau RC, McCormick HWS, Mowbray TJ (1998) Antipatterns: Refactoring Software, Architectures, and Projects in Crisis, 1 edn. Wiley, New York
- Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. *IEEE Trans Softw Eng* 20(6):476–493. <https://doi.org/10.1109/32.295895>
- Couto C, Pires P, Valente MT, da Silva Bigonha R, Hora AC, Anquetil N (2013) Bugmaps-granger: A tool for causality analysis between source code metrics and bugs
- Couto C, Pires P, Valente MT, Bigonha RS, Anquetil N (2014) Predicting software defects with causality tests. *J Syst Softw* 93:24–41. <https://doi.org/10.1016/j.jss.2014.01.033>. <http://www.sciencedirect.com/science/article/pii/S0164121214000351>
- Cox D, Miller H (1965) The Theory of Stochastic Process, 1 edn. Chapman and Hall, London
- Cramer H (1946) Mathematical Methods of Statistics, 1 edn. Princeton University Press
- Cui D, Liu T, Cai Y, Zheng Q, Feng Q, Jin W, Guo J, Qu Y (2019) Investigating the impact of multiple dependency structures on software defects. In: Proceedings of the 41st International Conference on Software Engineering, ICSE '19. IEEE Press, Piscataway, pp 584–595. <https://doi.org/10.1109/ICSE.2019.00069>
- de Andrade HS, Almeida E, Crnkovic I (2014) Architectural bad smells in software product lines: an exploratory study. In: Proceedings of the WICSA 2014 Companion Volume, WICSA '14 Companion. ACM, pp 12:1–12:6. <https://doi.org/10.1145/2578128.2578237>
- Deursen Av, Moonen L, Bergh Avd, Kok G (2001) Refactoring test code. In: M. Marchesi (ed.) Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001). University of Cagliari, pp 92–95
- Ferreira KA, Bigonha MA, Bigonha RS, Mendes LF, Almeida HC (2012) Identifying thresholds for object-oriented software metrics. *J Syst Softw* 85(2):244–257. <https://doi.org/10.1016/j.jss.2011.05.044>. <http://www.sciencedirect.com/science/article/pii/S0164121211001385>. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering
- Filó TGS, Da Silva Bigonha MA (2015) A catalogue of thresholds for object-oriented software metrics

- Fontana FA, Ferme V, Marino A, Walter B, Martenka P (2013) Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In: 2013 IEEE International conference on software maintenance (ICSM). IEEE, pp 260–269
- Fontana FA, Ferme V, Zanoni M (2015) Towards assessing software architecture quality by exploiting code smell relations. In: Proceedings of the 2015 IEEE/ACM 2Nd International Workshop on Software Architecture and Metrics, SAM '15. IEEE Computer Society, Washington, pp 1–7. <https://doi.org/10.1109/SAM.2015.8>
- Fontana FA, Pigazzini I, Roveda R, Zanoni M (2016) Automatic detection of instability architectural smells. In: 2016 IEEE international conference on Software maintenance and evolution (ICSME). IEEE, pp 433–437
- Fontana FA, Pigazzini I, Roveda R, Tamburri D, Zanoni M, Nitto ED (2017) Arcan: a tool for architectural smells detection. In: 2017 IEEE International conference on software architecture workshops (ICSAW). IEEE, pp 282–285
- Fontana FA, Lenarduzzi V, Roveda R, Taibi D (2019) Are architectural smells independent from code smells? an empirical study. J Syst Softw 154:139–156. <https://doi.org/10.1016/j.jss.2019.04.066>. <http://www.sciencedirect.com/science/article/pii/S0164121219301013>
- Fourati R, Bouassida N, Abdallah HB (2011) A Metric-Based Approach for Anti-pattern Detection in UML Designs. In: Computer and information science 2011. Springer, Berlin, pp 17–33
- Fowler M (1999) Refactoring: Improving the Design of Existing Programs, 1 edn. Addison-Wesley Professional
- Fuller WA (1976) Introduction to Statistical Time Series, 1 edn. Wiley, New York
- Garcia J, Popescu D, Edwards G, Medvidovic N (2009) Toward a catalogue of architectural bad smells. In: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems, QoSA '09. Springer, Berlin, pp 146–162. https://doi.org/10.1007/978-3-642-02351-4_10
- Garcia J (2014) Technical report: Architectural Smell Definitions and Formalizations. <http://csse.usc.edu/TECHRPTS/2014/reports/usc-csse-2014-500.pdf>. [Online; accessed 16-June-2017]
- Granger CWJ (1969) Investigating causal relations by econometric models and cross-spectral methods. Econometrica 37(3):424–438. <http://www.jstor.org/stable/1912791>
- Guimaraes E, Garcia A, Cai Y (2014) Exploring blueprints on the prioritization of architecturally relevant code anomalies – a controlled experiment. In: 2014 IEEE 38Th annual computer software and applications conference, pp 344–353. <https://doi.org/10.1109/COMPSAC.2014.57>
- Guimaraes E, Vidal S, Garcia A, Diaz Pace JA, Marcos C (2018) Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support. Softw Pract Exper 48(5):1077–1106. <https://doi.org/10.1002/spe.2563>
- Herbold S, Grabowski J, Waack S (2011) Calculation and optimization of thresholds for sets of software metrics. Empir Softw Eng 16:812–841
- Hochstein L, Lindvall M (2003) Diagnosing architectural degeneration. In: 2003. Proceedings of 28Th annual NASA goddard software engineering workshop, pp 137–142
- Kessentini W, Kessentini M, Sahraoui H, Bechikh S, Ouni A (2014) A cooperative parallel Search-Based software engineering approach for Code-Smells detection. IEEE Trans Softw Eng 40(9):841–861
- Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software, pp 305–314. IEEE Computer Society
- Koschke R (2007) Duplication, Redundancy, and Similarity in Software, no. 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl. <http://drops.dagstuhl.de/opus/volltexte/2007/962> Koschke R, Merlo E, Walenstein A (eds)
- Koziolek H, Domis D, Goldschmidt T, Vorst P (2013) Measuring architecture sustainability. IEEE Softw 30(6):54–62. <https://doi.org/10.1109/MS.2013.101>
- Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. IEEE Softw 29(6):18–21. <https://doi.org/10.1109/MS.2012.167>
- Kwiatkowski D, Phillips PC, Schmidt P, Shin Y (1992) Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? J Econ 54(1):159–178. [https://doi.org/10.1016/0304-4076\(92\)90104-Y](https://doi.org/10.1016/0304-4076(92)90104-Y). <http://www.sciencedirect.com/science/article/pii/030440769290104Y>
- Lanza M, Marinescu R, Ducasse S (2005) Object-Oriented Metrics in practice. Springer, Berlin
- Le DM, Carrillo C, Capilla R, Medvidovic N (2016) Relating architectural decay and sustainability of software systems. In: 2016 13Th working IEEE/IFIP conference on software architecture (WICSA), pp 178–181. <https://doi.org/10.1109/WICSA.2016.15>
- Le DM, Link D, Shahbazian A, Medvidovic N (2018) An empirical study of architectural decay in open-source software. In: 2018 IEEE International conference on software architecture (ICSA), pp 176–17609. <https://doi.org/10.1109/ICSA.2018.00027>

- Li Z, Liang P, Avgeriou P (2015) Architectural technical debt identification based on architecture decisions and change scenarios. In: 2015 12Th working IEEE/IFIP conference on software architecture, pp 65–74. <https://doi.org/10.1109/WICSA.2015.19>
- Lipper M, Roock S (2006) Refactoring in large software projects: performing complex restructurings successfully. Wiley, New York
- Liu H, Ma Z, Shao W, Niu Z (2012) Schedule of bad smell detection and resolution: a new way to save effort. *IEEE Trans Softw Eng* 38(1):220–235. <https://doi.org/10.1109/TSE.2011.9>
- Lozano A, Mens K, Portugal J (2015) Analyzing code evolution to uncover relations. In: 2015 IEEE 2Nd international workshop on patterns promotion and anti-patterns prevention (PPAP), pp 1–4. <https://doi.org/10.1109/PPAP.2015.7076847>
- Lutellier T, Chollak D, Garcia J, Tan L, Rasyide D, Medvidović N, Kroeger R (2018) Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Trans Softw Eng* 44(2):159–181. <https://doi.org/10.1109/TSE.2017.2671865>
- Ma W, Chen L, Zhou Y, Xu B, Zhou X (2015) Are anti-patterns coupled? an empirical study. In: 2015 IEEE International conference on software quality, reliability and security, pp 242–251. <https://doi.org/10.1109/QRS.2015.43>
- Macia I, Arcoverde R, Garcia A, Chavez C, von Staa A (2012a) On the relevance of code anomalies for identifying architecture degradation symptoms. In: 2012 16Th european conference on software maintenance and reengineering, pp 277–286
- Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A (2012b) Are automatically-detected code anomalies relevant to architectural modularity? In: the 11th annual international conference, pp 167–178 ACM Press
- Mantyla M, Vanhanen J, Lassenius C (2003a) A taxonomy and an initial empirical study of bad smells in code. In: 2003. ICSM 2003. Proceedings of International conference on software maintenance, pp 381–384. <https://doi.org/10.1109/ICSM.2003.1235447>
- Mantyla M, Vanhanen J, Lassenius C (2003b) A taxonomy and an initial empirical study of bad smells in code. In: 2003. ICSM 2003. Proceedings of International conference on software maintenance, pp 381–384. <https://doi.org/10.1109/ICSM.2003.1235447>
- Marinescu R (2004) Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, ICSM '04. IEEE Computer Society, pp 350–359
- Martin RC (2002) Agile software development: principles, patterns, and practices. Prentice Hall
- Martini A, Fontana FA, Biaggi A, Roveda R (2018) Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In: 12Th european conference on software architecture (ECSA 2018). <https://doi.org/10.1109/ICSA.2018.00027>
- Miller GA (1956) The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychol Rev* 63(2):81–97
- Mo R, Cai Y, Kazman R, Xiao L (2015) Hotspot patterns: The formal definition and automatic detection of architecture smells. In: WICSA. IEEE Computer Society, pp 51–60. <https://doi.org/10.1109/WICSA.2015.12>
- Mo R, Cai Y, Kazman R, Xiao L, Feng Q (2019) Architecture anti-patterns: Automatically detectable violations of design principles. *EEE Transactions on Software Engineering*:1–1. <https://doi.org/10.1109/TSE.2019.2910856>
- Moha N, Guéhéneuc Y, Duchien L, Meur AL (2010) DECOR: A Method for the specification and detection of code and design smells. *IEEE Trans Softw Eng* 36(1):20–36. <https://doi.org/10.1109/TSE.2009.50>
- Munaiyah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. *Empir Softw Eng* 22(6):3219–3253. <https://doi.org/10.1007/s10664-017-9512-6>
- Nam D, Lee YK, Medvidovic N (2018) Eva: a tool for visualizing software architectural evolution. In: 2018 IEEE/ACM 40Th international conference on software engineering: Companion (ICSE-companion), pp 53–56
- Oizumi WN, Garcia AF, Colanzi TE, Ferreira M, Staa A (2014) When code-anomaly agglomerations represent architectural problems? an exploratory study. In: 2014 Brazilian symposium on software engineering, pp 91–100. <https://doi.org/10.1109/SBES.2014.18>
- Oizumi WN, Garcia AF, Colanzi TE, Ferreira M, Staa AV (2015) On the relationship of code-anomaly agglomerations and architectural problems. *J Softw Eng Res Dev* 3(1):11. <https://doi.org/10.1186/s40411-015-0025-y>
- Oizumi W, Garcia A, da Silva Sousa L, Cafeo B, Zhao Y (2016) Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16. ACM, New York, pp 440–451. <https://doi.org/10.1145/2884781.2884868>

- Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015) Mining version histories for detecting code smells. *IEEE Trans Softw Eng* 41(5):462–489
- Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: 2013 28Th IEEE/ACM international conference on automated software engineering (ASE), pp 268–278. <https://doi.org/10.1109/ASE.2013.6693086>
- Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inf Softw Technol* 99:1–10. <https://doi.org/10.1016/j.infsof.2018.02.004>
- Pietrzak B, Walter B (2006) Leveraging code smell detection with inter-smell relations. In: Abrahamsson P, Marchesi M, Succi G (eds) Extreme programming and agile processes in software engineering. Springer, Berlin, pp 75–84
- Samarthyam G, Suryanarayana G, Sharma T (2016) Refactoring for software architecture smells. In: Proceedings of the 1st International Workshop on Software Refactoring. ACM, pp 1–4. <https://doi.org/10.1145/2975945.2975946>
- Sharma T, Fragkoulis M, Spinellis D (2016a) Does your configuration code smell?. In: Proceedings of the 13th International Workshop on Mining Software Repositories, MSR'16, pp 189–200. <https://doi.org/10.1145/2901739.2901761>
- Sharma T, Mishra P, Tiwari R (2016b) Designite — a software design quality assessment tool. In: Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities, BRIDGE '16. ACM. <https://doi.org/10.1145/2896935.2896938>
- Sharma T, Fragkoulis M, Spinellis D (2017) House of cards: Code smells in open-source c# repositories. In: 2017 ACM/IEEE International symposium on empirical software engineering and measurement (ESEM), pp 424–429. <https://doi.org/10.1109/ESEM.2017.57>
- Sharma T (2018a) Designite - A Software Design Quality Assessment Tool. <http://www.designite-tools.com>. [Online; accessed 31-Mar-2018]
- Sharma T, Spinellis D (2018b) A survey on software smells. *J Syst Softw* 138:158–173. <https://doi.org/10.1016/j.jss.2017.12.034>. <http://www.sciencedirect.com/science/article/pii/S0164121217303114>
- Sharma T (2019a) A dataset of code smells. <https://doi.org/10.5281/zenodo.2538646>
- Sharma T (2019b) Designite validation data. <https://doi.org/10.5281/zenodo.2538655>
- Suryanarayana G, Samarthym G, Sharma T (2014) Refactoring for Software Design Smells: Managing Technical Debt, 1 edn. Morgan Kaufmann
- Tamburri D, Kazman R, Van Den Heuvel WJ (2019) Splicing community and software architecture smells in agile teams: an industrial study. In: Agile and lean: organizations, Products and Development. <https://doi.org/10.24251/HICSS.2019.843>
- Tran JB, Godfrey MW, Lee EHS, Holt RC (2000) Architectural repair of open source software. In: Proceedings IWPC 2000. 8th International Workshop on Program Comprehension, pp 48–59. <https://doi.org/10.1109/WPC.2000.852479>
- VanderWeele TJ, Shpitser I (2013) On the definition of a confounder. *Ann Stat* 41(1):196–220. <https://doi.org/10.1214/12-aos1058>
- Verdecchia R, Malavolta I, Lago P (2018) Architectural technical debt identification: The research landscape. In: Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18. ACM, New York, pp 11–20. <https://doi.org/10.1145/3194164.3194176>
- Vidal S, Vazquez H, Díaz-pace JA, Marcos C, Garcia A, Oizumi W (2016a) JSpIRIT: A flexible tool for the analysis of code smells. In: Proceedings - International Conference of the Chilean Computer Science Society, SCCC. Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, IEEE, pp 1–6
- Vidal SA, Marcos C, Díaz-Pace JA (2016b) An approach to prioritize code smells for refactoring. *Autom Softw Eng* 23(3):501–532
- Walter B, Fontana FA, Ferme V (2018) Code smells and their collocations: a large-scale experiment on open-source systems. *J Syst Softw* 144:1–21
- Xiao L, Cai Y, Kazman R (2014) Titan: a toolset that connects software architecture with quality analysis. In: FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Drexel University, ACM, pp 763–766
- Yamashita A, Moonen L (2013a) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: 2013 35Th international conference on software engineering (ICSE). <https://doi.org/10.1109/ICSE.2013.6606614>, pp 682–691
- Yamashita A, Moonen L (2013b) To what extent can maintenance problems be predicted by code smell detection? – an empirical study. *Inf Softw Technol* 55(12):2223–2242
- Yamashita A, Zanoni M, Fontana FA, Walter B (2015) 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), pp 121–130

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Tushar Sharma is currently a Research Scientist at Siemens Corporate Technology, Charlotte, USA. The topics related to software design and architecture, refactoring, software quality (specifically code and design quality), and technical debt define his career interests. He earned PhD from the Athens University of Economics and Business, Athens, Greece with a specialization in software engineering in May 2019. Earlier, he obtained an MS in Computer Science from the Indian Institute of Technology-Madras, Chennai, India. His professional experience includes working with Siemens Research and Technology Center, Bangalore, India for more than seven years (2008–2015). He co-authored “Refactoring for Software Design Smells: Managing Technical Debt” and two Oracle Java certification books. He has founded and developed Designite which is a software design quality assessment tool used by many practitioners and researchers worldwide. He is an IEEE Senior Member.



Paramvir Singh is a Professional Teaching Fellow in the School of Computer Science at the University of Auckland. Prior to this, he worked as Assistant Professor in the Department of Computer Science and Engineering (CSE), National Institute of Technology, Jalandhar, India. He received his Ph.D. degree in CSE from Guru Nanak Dev University, Amritsar, India, in 2011. His research interests include software design and architecture, computer science education, dynamic program analysis, and evidence-based software engineering. He has around 50 research publications in several areas of computer science and software engineering, and has served as reviewer for software engineering research publication venues including ICSE, EASE, SANER, JSS, IST, among others. He is a member of the ACM, IEEE and the IEEE Computer Society, and a life member of ISTE.



Diomidis Spinellis is the Head of and a Professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Greece. His research interests include software engineering, IT security, and cloud systems engineering. He has written two award-winning, widely-translated books: “Code Reading” and “Code Quality: The Open Source Perspective.” His most recent book is Effective Debugging: 66 Specific Ways to Debug Software and Systems. Dr. Spinellis has also published more than 300 technical papers in journals and refereed conference proceedings, which have received more than 8000 citations. He served for a decade as a member of the IEEE Software editorial board, authoring the regular “Tools of the Trade” column, and as the magazine’s Editor-in-Chief over the period 2015’2018. He has contributed code that ships with Apple’s macOS and BSD Unix and is the developer of CScout, UMLGraph, dgsh, and other open-source software packages, libraries, and tools. He holds an MEng in Software Engineering and a PhD in Computer Science, both from Imperial College London. Dr. Spinellis has served as an elected member of the IEEE Computer Society Board of Governors (2013’015), and is a senior member of the ACM and the IEEE.