An Empirical Study of Architectural Decay in Open-Source Software

Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic
University of Southern California
Los Angeles, CA 90089, USA
{ducmle,dlink,armansha,neno}@usc.edu

***Abstract*—Architecture is the set of principal design decisions about a software system. In practice, new architectural decisions are added and existing ones reversed or modified throughout a system's lifetime. Frequently, these decisions deviate from the architect's well-considered intent, and software systems regularly exhibit increased architectural decay as they evolve. The manifestations of such ill-considered design decisions are seen as "architectural smells". To date, there has been no in-depth study of the characteristics or trends involving this phenomenon. Instead, when referring to architectural smells and their negative effects, both researchers and practitioners had to rely on folklore and their personal, inherently limited experience. In this paper, we report on the systematic step we have taken in investigating the nature and impact of architectural smells. We have selected a set of representative architectural smells from literature and analyzed their instances in 421 versions from 8 open-source software systems. We have (1) developed algorithms to automatically detect instances of multiple architectural smell types, and (2) analyzed relationships between the detected smells and the lists of issues reported in the systems' respective issue trackers. Our study shows that architectural smells have tangible negative consequences in the form of implementation issues as well as code commits requiring increased maintenance effort throughout a system's lifetime.**

I. Introduction

Software systems tend to experience increasing problems as they are modified during their lifetimes. In turn, this impacts key system qualities: correctness, performance, reliability, maintainability, and so on. Of particular interest to us is *architectural decay*, a class of problems caused by the introduction of design decisions that have not been carefully thought through by the system's engineers.

Although the downsides of architectural decay have been recognized from the very beginning of the study of software architecture [41], there has been limited research into this phenomenon. Prior work [22], [23], [25], [43], [32], [36] has analyzed the symptoms of architectural decay—*architectural smells* (e.g., scattered concerns among system components or circular dependencies between them)—and their negative impact. However, the authors of those studies have relied on their own experience and small case studies, making it hard to draw more general conclusions. Furthermore, the existing empirical studies have focused on code-level problems rather than architectural ones. Other prior work [15], [31], [37] has tried using code-level smells [17] as a means of detecting architectural problems. However, it has been shown that a code smell rarely indicates an architectural problem [38]. In response, researchers have tried to identify *groups* of code smells that can point to design problems, i.e., to design smells [37]. The results of these efforts have been instructive to our research. However, as defined, design smells tend to focus on problems and abstractions that are not necessarily architectural (e.g., broken inheritance hierarchy or duplicate class implementations with different APIs).

Considering that some of the ideas that underlie code smells also apply to architectural smells, it may be tempting to simply regard architectural smells as higher-abstraction relatives of code smells. However, having a smell at one level does not indicate that it exists at the other, neither by definition nor correlation. Second, while code-level and design smells may be solvable without any knowledge of the architecture, e.g., by employing better programming practices, this is not the case with architectural smells.

Figure 1 illustrates the absence of correlation between code- and architecture-level smells, with an example of Dependency Cycle (DC). A DC smell occurs when the directed interconnections between two or more elements form a cycle. Figure 1 shows three different cases of a software architecture made up of two components, C_1 and C_2. Each component contains two implementation-level entities. The arrows represent dependencies among entities. In sub-figure (i), there is a DC at the code-level between entities 1 and 3, which also forms a DC at the architecture-level between C_1 and C_2. In sub-figure (ii), the code-level DC between entities 1 and 2 resides completely within C_1, and there is no architecture-level DC between C_1 and C_2. Lastly, in sub-figure (iii), there is no DC at the code-level, but C_1 and C_2 are still affected by an architecture-level DC, formed by two dependencies, from entity 1 to entity 3 and from entity 4 to entity 2.

This relationship (or, rather, lack thereof) calls for an in-depth study of architectural decay that uses *architectural* smells—rather than code or design smells—as decay manifestations. This paper reports on such a study, in which we analyzed existing systems for the emergence, proliferation, and possible elimination of architectural smells over time. Recent advances in two areas provided a foundation for our empirical study. First, to identify architectural smells, a system's *actual* architecture—as opposed to its idealized architecture—must be recovered from the implementation. A recent study of architecture recovery techniques [20] identified the scenarios in which various available techniques are accurate and scalable. Second, in order to identify symptoms of architectural decay, researchers have collected a growing catalog of architectural smells [30], [23], [22], and have shown the usefulness of certain smells in highlighting different problems in real systems [32], [36].

A direct motivation for our study was the frequently-repeated, but to date empirically unsupported, claim that architectural smells are detrimental to a system. To validate this claim, in addition to the code repositories, we rely on issue repositories, in which system stakeholders report bugs, describe perfective or adaptive changes, discuss re-engineering the system, etc. Specifically, we have (1) selected and developed algorithms to automatically detect instances of multiple representative architectural smell types, and (2) mined the reported issues to gauge the impact of these architectural smells. Two key observations have guided this work: (a) If relationships between architectural decay and implementation issues exist, studying these relationships can help us uncover the architectural root causes of implementation problems. (b) Architectural smells may be used to pinpoint the issue-prone parts of a software system even before stakeholders bring up the issues.

To this end, we conducted the largest empirical study to date of architectural decay in long-lived software systems. Our study's scope is reflected in the total numbers of versions (421) of 8 subject systems comprising 376 MSLOC, examined implementation issues (41,889), identified architectural smells (172,934) spanning 6 different smell types, applied architecture-recovery techniques (3) resulting in distinct architectural views produced for each system, and analyzed architectural models (1,263, i.e., three views per system version).

Our study empirically demonstrates a finding that had previously only been suspected: The parts of a system that exhibit signs of architectural decay experience more implementation-level problems than the architecturally "clean" parts. For example, we find that files that implement "smelly" parts of a system's architecture are more issue-prone and change-prone than the "clean" files. We also observed an increase over time of reported issues and maintenance effort related to long-lived architectural smells. Our findings indicate that underlying architectural problems have increasing negative consequences on the quality of a system's implementation, and may be the root cause of many implementation-level issues.

The remainder of the paper is organized as follows. Section II summarizes the related research. Section III describes the selected architectural smells and their detection algorithms. Section IV describes our research hypotheses and the setup for our empirical study. Sections V and VI present our key results, and discuss validity threats. Related work and conclusions round out the paper. Unless otherwise noted, we will refer to "architectural smell" simply as "smell" in the rest of the paper.

II. Foundation

Our work discussed in this paper is directly enabled by three research threads: (1) software architecture recovery, (2) definition of architectural smells, and (3) tracking of implementation issues. We overview each of these threads.

A. Architecture Recovery

In this work, an implemented software system's architecture is represented as a graph that captures components as vertices and their interconnections (e.g., call dependencies and logical couplings) as edges. Each component, in turn, contains a set of entities (e.g., implementation classes).

Our previous studies [20], [27] have shown that three architecture recovery techniques— *ACDC* [48], *ARC* [24], and *PKG* [27]—generally exhibit higher accuracy than their competitors. For this reason, we use these three techniques in our study of architectural decay. *ACDC* is oriented toward components that are based on structural patterns (e.g., components consisting of entities that together form a particular subgraph). *ARC* produces components that are semantically coherent due to sharing similar system-level concerns (e.g., a component whose main concern is handling distributed jobs). Finally, a system's package structure extracted by *PKG* forms a sort of "developers' perception" of the as-implemented architecture [27].

While the three selected recovery methods differ in how they approach architecture, they all produce (1) clusters of source-code entities, (2) dependencies among code entities within a cluster, and (3) dependencies across clusters. We use this information to detect architectural smells.

B. Architectural Smells

Similar to the concept of smells at other levels of system abstraction (namely, code and design smells), architectural smells are instances of poor design decisions [34] — at the architectural level. They negatively impact system lifecycle properties, such as understandability, testability, extensibility, and reusability [22]. While code smells [17], anti-patterns [11], or structural design smells [19] originate from implementation constructs (e.g., classes), architectural smells stem from poor use of software architecture-level abstractions — components, connectors, interfaces, patterns, styles, etc. Detected instances of architectural smells are candidates for restructuring [9], to help prevent architectural decay and improve system quality.

Researchers have collected and reported a growing catalog of architectural smells. Garcia et al. [22], [23] have identified

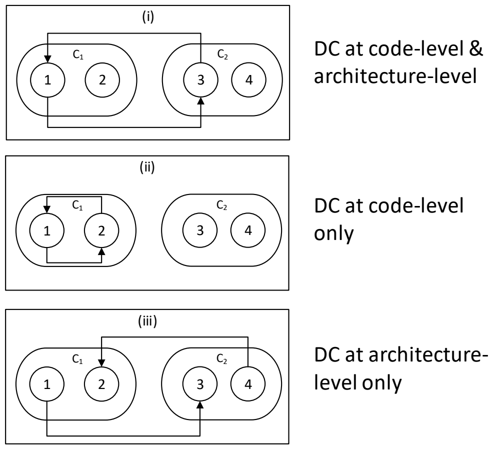

(i)
C_1
C_2
1
2
3
4
DC at code-level & architecture-level
(ii)
DC at code-level only
(iii)
DC at architecture-level only

FIG. 1: DEPENDENCY CYCLE.

an initial set of four architectural smells related to connectors, interfaces, and concerns. Mo et al. [36] extended Garcia's list with a new concern-related smell. Ganesh et al. [19] also summarized a catalog of structural design smells, some of which are actually at the architecture-level. We recently provided a short description of 12 different architectural smells [28] and proposed a framework for relating architectural smells to system sustainability. The work described in this paper extends these prior approaches.

C. Issue Tracking Systems

All subject systems that were selected for this paper use Jira [1] as their issue repository. Our approach relates implementation issues with smells based on the data available on Jira. A similar approach can be applied to other repositories.

When reporting implementation issues, engineers categorize them into different types: *bug*, *new feature*, feature *improvement*, *task* to be performed, etc. Each issue has a status that indicates where the issue is in its lifecycle [5]. Each issue starts as "open", progresses to "resolved", and finally to "closed". We restrict our study to closed and fixed issues because they were verified and addressed by developers, so that any effects caused by them would presumably appear in certain system versions and disappear once the issue is addressed. Additionally, a fixed issue contains information that is useful for our study: (1) *affected versions* in which the issue has been found, (2) *type* of the issue, and (3) *fixing commits*, i.e., the changes applied to the system to resolve the issue. Finding fixing commits is not always easy since there is no standard method for engineers to keep track of this information on issues trackers. In Jira, we found three popular methods: (1) directly mapping to fixing commits, (2) using pull requests, and (3) using patch files. Our approach supports all three methods.

Based on the collected information, issues are mapped to detected smells using the model depicted in Figure 2. First, we find the system versions that the issue affects. Then we find the smells present in those versions. We say the issue is infected by a given smell iff (1) both the issue and the smell affect the same version of a system and (2) the resolution of the issue changes files that are involved in the smell. Note that resolving the issue may or may not remove the smell that may have caused the issue in the first place: developers may find a different workaround. Based on this relationship between issues and smells, we studied if the characteristics of an issue (e.g. type, number of fixing commits) are correlated with whether the issue is infected by a given smell.

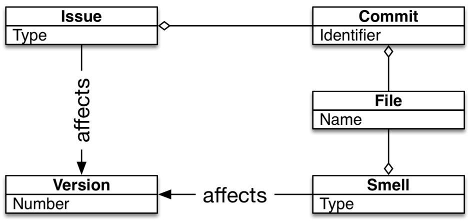


FIG. 2: MAPPING ARCHITECTURAL SMELLS TO ISSUES.

III. SELECTING, CAPTURING, AND DETECTING SMELLS

To ensure that the architectural smells included in this study are important and representative, we carefully considered the existing literature to find aspects of architecture that are important to engineers. These include dependencies and interfaces of components as fundamental architectural building blocks [41], couplings as root causes of architectural debt [49], concerns extracted from code to help engineers understand the responsibilities of components [7], [33], etc. We reviewed existing collections of architectural smells [22], [36], [23], [30], leveraged the foundational work on architectural patterns and styles, as well code- and design-level smells. Our selection of smells was also guided by key software engineering principles (e.g., separation of concerns [13], modularity [40]). Finally, to be practically usable, especially on large, multi-version systems, the detection algorithms of selected smells must be executable automatically or with minimal human assistance.

Prior work [22], [23], [36], [28], [30] identified 17 architectural smells. Deeper analysis revealed that three pairs of smells were essentially duplicates. From the remaining list of 14, we removed two smells because they are related to software connectors [22]: since none of the architecture recovery techniques we rely on in this work identify explicit connectors, we are unable to detect connector-based smells automatically. One additional smell (Ambiguous Interface [30]) was removed because its detection requires significant human involvement. We then conducted a preliminary study of the remaining 11 smells in the context of eight open-source systems from the Apache Software Foundation (Table I). We found that six of the architectural smells appeared frequently in those systems. Consequently, we settled on these six smells, which cover four important aspects of software architecture: concerns, dependencies, interfaces, and couplings. These six smells are *Concern Overload*, *Dependency Cycle*, *Link Overload*, *Unused Interface*, *Sloppy Delegation*, and *Co-change Coupling*.

To provide a consistent treatment of smells and their detection algorithms, we will first formalize architectural concepts in Section III-A. Different approaches have emerged over the past two decades to formalize software architecture [6], [46], [35]. We extend the formalization from Mo et al. [36] because it is easy to understand and was created to capture the concept of architectural decay. Based on this formalization, the formal definitions of and detection algorithms for the six selected architectural smells will be provided in Section II-B.

A. Formalization of Architectural Concepts

Figure 3 shows a notional software architecture A that comprises two components, C_1 and C_2. Each component contains

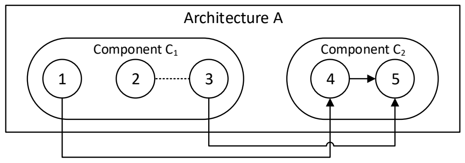


FIG. 3: A NOTIONAL SOFTWARE SYSTEM'S ARCHITECTURE.

multiple implementation-level entities. Between entities, links are presented by solid arrows and couplings by dashed lines. In the view of typical recovery methods [21], we represent the structure of a system's architecture as a graph A whose vertices represent the system's components C and topology represents the connections embodied in the set of links L and the set of couplings Cp between these components: $A = (C, L, Cp)$

Since our study involves a concern-based recovery technique (ARC), we will also formalize this concept. For our purposes, the architecture of a software system can be associated with a nonempty set of topics T, which captures functionalities of the system and its components. We define a topic as a probability distribution Pd over the system's nonempty set of keywords W, whose elements are used to "describe" that system (e.g., via comments in source code). By examining the words that have the highest probabilities in a topic, the meaning of that topic may be discerned. In this way, a topic can serve as a representation of a *concern* addressed by a system. The set of topics T is then a representation of the system's concerns.

$$W = \{w_i \mid i \in \mathbb{N}\} \quad T = \{z_i \mid i \in \mathbb{N}\} \quad z = Pd(W)$$

A component is a tuple comprising the component's internal entities E and the probability distribution θ over the system's topics T. Entities are implementation elements used to build a system. An entity e contains its interface I, a set of links L_E, and a set of couplings Cp_E to other entities. In OO systems, entities are classes and interfaces are public methods.

$$C = \{c_i \mid i \in \mathbb{N}\} \quad c = (E, \theta_c)$$
$$E = \{e_i \mid i \in \mathbb{N}\} \quad e_i = (I_i, L_{e_i}, Cp_{e_i}) \quad I = \{ie_i \mid i \in \mathbb{N}\}$$

Both a link l and a coupling cp consist of a source src and a destination dst, which are the components' internal entities involved in an interconnection. Links are *unidirectional*, while couplings are *bidirectional*. The union of the links of all entities is the set of links L of the graph A. The union of the couplings of all entities is the set of couplings Cp of the graph A.

$$L = \cap_{i=1}^{n} L_{e_i} \quad L_e = \{l_j \mid i \in \mathbb{N}_0\} \quad l = (src, dst)$$
$$Cp = \cap_{i=1}^{n} Cp_{e_i} \quad Cp_e = \{cp_j \mid i \in \mathbb{N}_0\} \quad cp = (src, dst)$$

B. Smell Formalization and Detection Algorithms

One critical issue in smell detection is setting thresholds, i.e., defining the criteria that serve as indicators of smells. We set thresholds by using Interquartile analysis [47], which is a widely used, efficient technique [26] for detecting outliers (i.e., smells in our study) in a population without requiring it to have a normal probability distribution.

In the Interquartile method, the lower quartile (q_1) is the 25^{th} percentile, and the upper quartile (q_3) is the 75^{th} percentile of the data. The interquartile range (iqr) is defined as the interval between q_1 and q_3. $q_1 - (1.5 * iqr)$ and $q_3 + (1.5 * iqr)$ are defined as inner fences, that mark off the "reasonable" values from the outliers [14]. In this section, we will use a shorthand function ***getHighThreshold()***, which accepts a list of values and return the high value of the "inner fences".

1) Concern Overload (CO): indicates that a component implements an excessive number of concerns [13]. CO may increase the size of a component, hurting its maintainability. Formally, a component c suffers from this smell iff

Algorithm 1: detectCO

```
    Input: $C$: a set of components, $T$: a set of system concerns
    Output: $COsmells$ : a set of Component Concern Overload instances
 1  $COsmells \leftarrow \emptyset$
 2  $componentConcernCounts \leftarrow$ initialize all brick concern counts to 0
 3  for $c \in C$ do
 4      $T_c \leftarrow getConcernsOfComponent(c)$
 5      $th_{z_c} \leftarrow getHighThreshold(P(T_c))$
 6      for $z \in T_c$ do
 7          if $P(z \mid c) > th_{z_c}$ then
 8              $componentConcernCounts[c] = componentConcernCounts[c] + 1$
 9  $th_{co} \leftarrow getHighThreshold(componentConcernCounts)$
10  for $c \in C$ do
11      if $componentConcernCounts[c] > th_{co}$ then
12          $COsmells \leftarrow COsmells \cup \{c\}$
```

$$\mid \{z_j \mid (j \in \mathbb{N}) \wedge (P(z_j \mid c) > th\_z_c)\} \mid > th_{co}$$

where thresholds $0 \le th\_z_c \le 1$ and $th_{co} \in \mathbb{N}$, respectively, indicate a topic is significantly represented in the component and the maximum acceptable number of concerns per component.

Algorithm 1, **detectCO**, determines which components in the system have CO. **detectCO** begins by creating a map, *componentConcernCounts*, where keys are components and values are numbers of relevant concerns in each component (Lines 3-8). While creating the map, threshold th_{z_c} is dynamically computed (Line 5) and used to determine prevalent concerns for each component.**detectCO** uses that map to compute threshold th_{co} (Line 9), which is then used to determine which components have the CO smell (Lines 10-12).

2) Dependency Cycle (DC): indicates a set of components whose links form a circular chain, causing changes to one component to possibly affect all other components in the cycle. Such high coupling between components violates the principle of modularity. Formally, this smell occurs in a set of three or more components iff

$$\begin{aligned} &\exists l \in L \mid (\forall x \mid (1 \le x \le k) \mid \\ &\quad ((x < k) \implies (l.src \in c_x.E \wedge l.dst \in c_{x+1}.E)) \wedge \\ &\quad ((x = k) \implies (l.src \in c_x.E \wedge l.dst \in c_1.E)) \end{aligned}$$

We detect DC smells by identifying *strongly connected subgraphs* in a system's architectural graph $G = (C, L)$. A strongly connected subgraph is one where each vertex is reachable from every other vertex. Algorithms that detect strongly connected subgraphs are well known [12] and can be used to identify DC smells.

3) Link Overload (LO): is a dependency-based smell that occurs when a component has interfaces involved in an excessive number of links (e.g., call dependencies), affecting the system's separation of concerns and isolation of changes. Formally, a component c suffers from both incoming and outgoing link overload iff

$$\mid \{l \in L \mid l.src \in c.E\} \mid + \mid \{l \in De \mid l.dst \in c.E\} \mid > th_{lo}$$

where th_{lo} is a threshold indicating the maximum number of links for a component that is considered reasonable.

Algorithm 2, **detectLO**, extracts the LO variants for a set of components C by examining their links L. The algorithm first determines the number of incoming and outgoing links per component (Lines 4-6). **detectLO** sets the threshold th_{lo} for each variant of LO (Lines 7-8). Finally, **detectLO** identifies

Algorithm 2: detectLO

```
Input: $C$: a set of components, $L$: links between components
Output: $LOsmells$ : a set of Link Overload instances
1  $LOsmells \leftarrow \emptyset$
2  $numLinks \leftarrow$ initialize map as empty
3  $directionality \leftarrow \{$"in","out","both"$\}$
4  for $c \in C$ do
5      for $d \in directionality$ do
6          $numLinks[(c,d)] \leftarrow numLinks[(c,d)] + getNumLinks(c,d,L)$
7  for $d \in directionality$ do
8      $th_{lo}[d] \leftarrow getHighThreshold(numLinks, d)$
9  for $c \in C$ do
10     for $d \in directionality$ do
11         if $getNumLinks(c,d,L) > th_{lo}[d]$ then
12             $LOsmells \leftarrow LOsmells \cup \{(c,d)\}$
```

each component and the directionality that indicates the variant of LO from which the component suffers (Lines 9-12).

4) Unused Interface (UI): is an interface of a system entity that is linked to no other entities. Including entities in a system without any associated use cases violates the principle of incremental development [18]. Having that unused entity adds unnecessary complexity to the component and the software system which, in turn, hinders maintenance. Formally, a component $c \in C$ contains a UI smell in entity $e \in b.E$ iff

$$(|e.I| \neq 0) \wedge (\nexists l \in e.L \mid l.dst = e)$$

Algorithm 3, **detectUI**, detects this smell. **detectUI** uses the set of links L to determine if an interface has been used. The algorithm checks every entity in each component (Lines 2-5), and if an entity has a public interface but no link, the entity and its parent component are added to the UI instances list.

Algorithm 3: detectUI

```
Input: $C$: a set of bricks, $L$: links between components
Output: $UIsmells$ : a set of Unused Interface instances
1  $UIsmells \leftarrow \emptyset$, $UCsmells \leftarrow \emptyset$
2  for $c \in C$ do
3      for $e \in c.E$ do
4          if $getNumInterfaces(e.I) > 0 \wedge getNumLinks(e.L) = 0$ then
5              $UIsmells \leftarrow UIsmells \cup \{(c,e)\}$
```

5) Sloppy Delegation (SD): occurs when a component delegates to other components functionality it could have performed internally. An example of SD is a component that manages all aspects of an aircraft's current velocity, fuel level, and altitude, but passes that data to an entity in another component that solely calculates that aircraft's burn rate. Such inappropriate separation of concerns complicates the system's data- and control-flow which, in turn, impacts system maintenance. Formally, SD occurs between components $c_1, c_2 \in C$ iff

$$\exists l \in L \mid l.src = e_1 \in c_1.E \wedge l.dst = e_2 \in c_2.E$$
$$(outLink(e2) = 0) \wedge (inLink(e2) < th_{sd}) \wedge (c_1 \not\equiv c_2)$$

where *outLink(e)* and *inLink(e)* return the numbers of links from and to entity e, respectively. Threshold th_{sd} ensures entity e_2 is not a library-type entity. In a strict constraint, $th_{sd} = 2$, meaning that e_2's functionality is only used by e_1.

Algorithm 4, **detectSD**, requires a threshold th_{sd}, which defines the minimum number of in-links to consider a delegation appropriate. The algorithm checks every link in each entity

Algorithm 4: detectSD

```
Input: $C$: a set of bricks, $L$: links between components,
       $th_{sd}$: threshold for relevance delegation
Output: $smells$ : a set of Sloppy Delegation instances
1  $smells \leftarrow \emptyset$
2  for $c_1 \in C$ do
3      for $e_1 \in c_1.E$ do
4          for $l \in e_1.L$ do
5              if $l.src = e_1$ then
6                  $e_2 \leftarrow l.dst$
7                  $c_2 \leftarrow getParent(e_2)$
8                  if $(e_1 \not\equiv e_2) \wedge (getOutLink(e_2) = 0) \wedge (getInLink(e_2) < th_{sd})$ then
9                      $smells \leftarrow smells \cup \{((c_1,e_1),(c_2,e_2))\}$
```

(Lines 2-5), and if a link has a *dst* entity which satisfies the checking condition of SD, then the *dst* entity and its parent component are added to the list of SD instances (Line 9).

6) Co-change Coupling (CC): is a logical coupling that occurs when changes to an entity of a given component also require changes to an entity in another component. Formally, a component $c \in C$ has a CC iff

$$e_i \in c.E \mid \sum_{i=1}^{n} |e_i.Cp| > th_{cc}$$

where th_{cc} specifies a threshold for an excessively high number of logical couplings.

Algorithm 5, **detectCC**, shows how to detect this type of smell. The algorithm first creates a map between components and their total numbers of co-changes (Lines 3-5). The *getNumCp* function returns the number of co-changes of an entity. **detectCC** uses these maps to compute a threshold, th_{cc}, which is the high inner-fence value (Line 6). Finally, the algorithm visits each component again, checks, and adds detected smell instances into the smell list (Lines 7-9).

Algorithm 5: detectCC

```
Input: $C$: a set of bricks, $Cp$: couplings between components
Output: $CCsmells$ :a set of Co-change Coupling instances
1  $CCsmells \leftarrow \emptyset$,
2  $numCp \leftarrow$ initialize map as empty
3  for $c \in C$ do
4      for $e \in c.E$ do
5          $numCp[c] \leftarrow numCp[c] + getNumCp(e.Cp)$
6  $th_{cc} \leftarrow getHighThreshold(numCp.values)$
7  for $c \in C$ do
8      if $numCp[c] > th_{cc}$ then
9          $CCsmells \leftarrow CCsmells \cup \{(c)\}$
```

IV. Empirical Study Setup

This section introduces three cornerstones of our empirical study. Our overarching research question and the hypotheses formulated to answer that question are described in Section IV-A. The selected subject systems and our selection criteria are presented in Section IV-B. Lastly, we describe how we adapted an existing tool suite for this study in Section IV-C.

A. Research Question and Hypotheses

Research Question: *How do architectural smells manifest themselves in a system's implementation?*

Our study seeks empirical evidence to support the long-standing claims about the negative impact of architectural decay on software systems. Specifically, as discussed above, we focus on exploring relationships between detected architectural smells and reported implementation issues. A correlation between the two will be an initial but important indicator that decay in a system's architecture can lead to tangible problems experienced by developers. Smells would help to point out the "hot spots" [3] in a system, i.e., highly active areas in the code that (1) have many bugs, (2) are involved in performance fine-tuning, or (3) serve as plug-in points for new features. Conversely, reported issues may be manifestations of underlying smells.

Based on the mapping between smells and issues described in Section II-C and Figure 2, we have defined several concepts that serve to formulate and validate our research hypotheses.

- We call a file "smelly" in a system version if that file is affected by at least one smell found in the recovered architectures of that version. Otherwise, the file is "clean".
- We call an issue "smelly" if at least one smelly file of a version affected by that issue is involved in the issue's resolution. If no such file exists, the issue is "clean".

To answer our research question in this study, we defined the following two hypotheses.

Hypothesis H1 (issue-proneness): *Smelly files are more likely to have associated issues than clean files.*
Hypothesis H2 (change-proneness): *Smelly files are more likely to be changed than clean files.*

The second hypothesis targets the frequent assumption about architectural decay — that it causes more maintenance effort.

B. Subject Systems

Our study reports the empirical results involving a set of Apache open-source projects. We selected Apache because it has well-maintained code repositories, release notes, and bug trackers. Table I shows the list of systems used in the study. To ensure the generality of our conclusions, we analyzed all available Apache systems that rely on Jira [1] to track issues, and used the following selection criteria:

1) *Different software domains*, to ensure broad applicability of our results.
2) *Tracking of issues and their fixing commits*, to help map architectural smells to implementation problems. Specifically, we analyze "resolved" and "closed" issues because they have complete sets of fixing commits.
3) *Large numbers of resolved and closed issues*, to give us sufficient data to ensure the accuracy of our analysis.
4) *Availability of multiple versions through the system's lifetime*, to allow tracking of architectural decay trends.

C. Tool Support for the Empirical Study

Analyzing and correlating all of the data displayed in Table I requires automated tool support. To that end, we have extended our architecture recovery and analysis tool suite, ARCADE, which is a state-of-the-art architecture recovery workbench [27]. Here, we highlight only the details of ARCADE relevant to our study. Figure 4 depicts specifically the workflow used in our study. We employ *Recovery Techniques* to extract architectures from the source code of subject systems. Three architectural views (*ACDC* [48], *ARC* [24], and *PKG* [27]) are extracted from each version of each system. *Smell Detector* implements the smell detection algorithms described in Section III-B. Once we extract issues from the Jira issue repository, *Smell-Issue Mapper* determines mappings between the issues and smells, as detailed in Section II-C and Figure 2. The mappings help us classify issues into "smelly" and "clean". Finally, *Analyzer* collects relevant data from the issues and smells, and applies statistical tests to verify our research hypotheses. ARCADE and subject systems in this paper are available for download from [2].

TABLE I: SUBJECT SYSTEMS ANALYZED IN OUR STUDY

System	**Domain**	**# Versions**	**# Issues**	**Avg. LOC**
Camel	Integration F-work	78	9665	1.13M
Continuum	Integration Server	11	2312	463K
CXF	Service F-work	120	6371	915K
Hadoop	Data Proc. F-work	63	9381	1.96M
Nutch	Web Crawler	21	1928	118K
OpenJPA	Java Persist.	20	1937	511K
Struts2	Web App F-work	36	4207	379K
Wicket	Web App F-work	72	6098	332K

V. EMPIRICAL STUDY RESULTS

Table II shows the average numbers of detected smells in each category, across all analyzed subject system versions, for each of the three architectural views produced by the employed recovery techniques (ACDC, ARC, and PKG). For each of our two research hypotheses, we discuss the method employed in attempting to validate it and the associated findings.

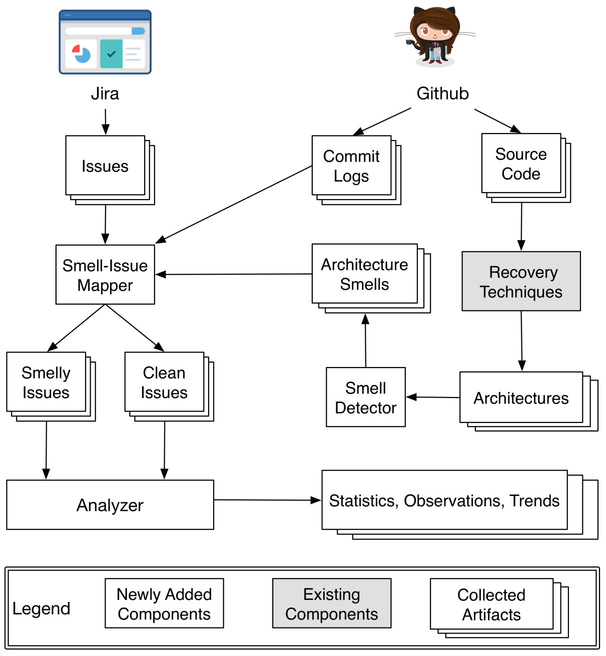


FIG. 4: ARCADE'S KEY COMPONENTS AND THE ARTIFACTS IT USES AND PRODUCES.

TABLE II: AVERAGE NUMBERS OF ARCHITECTURAL SMELLS PER SYSTEM VERSION
(NOTE: ACDC AND PKG ARE UNABLE TO IDENTIFY CONCERN OVERLOAD (CO) AS THEY DO NOT CAPTURE SYSTEM CONCERNS)

Arch. View →	*ACDC*						*ARC*						*PKG*					
System ↓	CO	DC	LO	UI	SD	CC	CO	DC	LO	UI	SD	CC	CO	DC	LO	UI	SD	CC
Camel	-	2	7	4	43	0	195	2	19	149	257	0	-	2	8	36	51	0
Continuum	-	1	2	1	7	3	9	1	3	11	14	4	-	1	3	0.5	14	2
CXF	-	3	31	10	107	64	78	2	13	41	109	80	-	3	53	67	139	65
Hadoop	-	1	5	20	26	18	27	1	42	35	77	47	-	1	8	18	33	18
Nutch	-	1	4	2	17	3	12	1	11	6	25	0	-	1	5	7	20	0
OpenJPA	-	1	8	3	33	0	19	1	18	7	38	0	-	1	11	4	10	0
Struts2	-	1	2	6	19	6	17	1	16	11	59	7	-	1	5	11	30	7
Wicket	-	2	9	4	68	0	81	2	26	53	266	0	-	2	13	22	88	0

TABLE III: AVERAGE NUMBERS OF ISSUES PER FILE

Arch. View →	*ACDC*				*ARC*				*PKG*			
System ↓	Smelly	Clean	Factor	p-value	Smelly	Clean	Factor	p-value	Smelly	Clean	Factor	p-value
Camel	2.5	1.7	1.47x	0.0001	2.5	1.4	1.71x	0.0001	2.5	1.6	1.56x	0.0001
Continuum	2.4	1.4	1.71x	0.0001	2.2	1.5	1.47x	0.0010	2.4	1.6	1.50x	0.0010
CXF	4.9	2.9	1.69x	0.0001	5.6	2.7	2.07x	0.0001	5.6	3.0	1.86x	0.0001
Hadoop	4.4	2.5	1.76x	0.0001	4.4	2.1	2.10x	0.0001	4.5	2.4	1.86x	0.0001
Nutch	3.8	1.6	2.38x	0.0001	3.8	1.7	2.24x	0.0001	3.9	1.6	2.44x	0.0001
OpenJPA	3.3	2.0	1.65x	0.0001	3.3	2.2	1.5x	0.0001	3.3	2.1	1.57x	0.0001
Struts2	2.1	1.7	1.24x	0.0020	2.1	1.7	1.24x	0.0100	2.1	1.7	1.24x	0.0016
Wicket	3.4	2.2	1.54x	0.0001	3.5	1.8	1.94x	0.0001	3.4	2.3	1.47x	0.0001

TABLE IV: AVERAGE NUMBERS OF COMMITS PER FILE

Arch. View →	*ACDC*				*ARC*				*PKG*			
System ↓	Smelly	Clean	Factor	p-value	Smelly	Clean	Factor	p-value	Smelly	Clean	Factor	p-value
Camel	14.7	11.3	1.30x	0.0001	13.7	8.9	1.53x	0.0001	14.3	9.7	1.47x	0.0001
Continuum	32.4	17.7	1.83x	0.0380	30.3	17.7	1.71x	0.0034	30.1	17.8	1.69x	0.0301
CXF	13.3	12.1	1.10x	0.0349	12.6	11.1	1.13x	0.0063	13.3	11.7	1.14x	0.0065
Hadoop	10.1	5.9	1.71x	0.0001	7.2	5.9	1.22x	0.1366	9.7	5.9	1.64x	0.0003
Nutch	11.9	9.3	1.27x	0.0015	12.3	9.3	1.32x	0.0005	11.8	9.3	1.26x	0.0024
OpenJPA	19.1	13.6	1.40x	0.0001	20.3	13.6	1.49x	0.0001	19.2	13.6	1.41x	0.0001
Struts2	17.6	9.7	1.81x	0.0001	17.9	9.8	1.83x	0.0001	17.7	9.7	1.82x	0.0001
Wicket	7.4	5.2	1.42x	0.0001	7.1	6.3	1.12x	0.0001	7.4	5.8	1.26x	0.0001

A. Hypothesis H1 – Issue-Proneness of Files

For each version of a software system, we first collected the files that were changed in order to fix the issues that affected that version. We then divided the collected files into two groups: smelly and clean. For each file, we counted the number of issues that affected it. Finally, we applied a *2-sample t-test* [39], which compares two population means, to find the difference in the number of issues between two groups. We used an alternative formulation of hypothesis *H1* to apply this test, namely, $(m_{smelly} - m_{clean}) > 0$.

Table III shows the average numbers of issues across all of the analyzed versions of each subject system under the three selected architectural views. The columns under each view show the averages numbers of issues in smelly and clean files, the difference factor between those two averages, and the *p-value* of the test to verify our hypothesis. We found that *p-value* < 0.05 for each of the three architectural views of each subject system. Therefore, we can accept the alternative formulation of the hypothesis, i.e., issue-proneness is strongly correlated with architectural smells. Our conclusion is statistically significant with a confidence level of 95% in all subject systems.

Multiplication factors in Table III range from 1.24 to 2.10, meaning that the issue rates in smelly files increase 24%–110% over clean files. The results of ARC recoveries have higher multiplication factors than *ACDC* and *PKG*. This suggests that smells detected under the ARC view may isolate issue-prone parts of a system better than smells detected under ACDC and PKG. We are currently investigating this observation further.

B. Hypothesis H2 – Change-Proneness of Files

To validate this hypothesis, we performed an analysis that is similar to the one of hypothesis *H1*. However, instead of counting the numbers of issues, we used git-log [4] to extract the numbers of commits related to smelly and clean files from the code repositories of the subject systems. We employed the same statistical method as in *H1* to find the difference between the two groups. We used an alternative formulation of hypothesis *H2* to apply this test, namely, $(m_{smelly} - m_{clean}) > 0$.

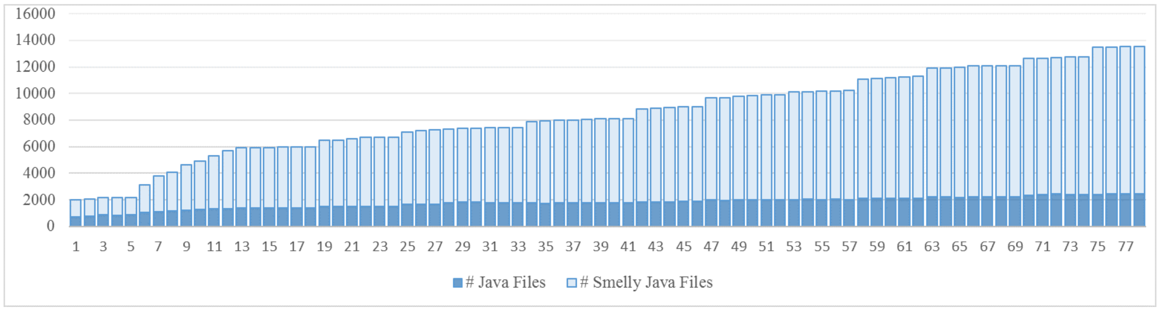


FIG. 5: PERCENTAGE OF SMELLY FILES IN CAMEL (ACDC VIEW).
SYSTEM VERSIONS ARE SHOWN ALONG THE BOTTOM; NUMBERS OF SYSTEM FILES ARE ON THE LEFT.

Table IV shows the average numbers of commits for the two groups across all of the analyzed versions of each subject system. The structure of this table is similar to Table III. With the exception of a single case (out of a total of 24 cases), we found that *p-value* < 0.05 in every subject system and across all architectural views. Therefore, we can accept the alternative hypothesis for 23 cases, i.e., the average number of changes in smelly files is higher than the analogous number in clean files. Our conclusion is statistically significant with a confidence level of 95% in those cases.

The exceptional case is Hadoop under the ARC view (*p-value* $= 0.1366$). Recall that ARC can be used to detect concern-based smells, while ACDC and PKG cannot. We found that many of Hadoop's issues are affected by the Concern Overload (CO) smell type: 385 affected issues out of the total 9,381 issues. This is a significantly greater proportion than in other systems (e.g., 34 affected issues out of 4,207 issues in Struts2). The average number of commits of smelly files (7.2) under the ARC view is lower compared to ACDC (10.1) and PKG (9.7) views. This hints that CO may affect files that have fewer commits in Hadoop. More generally, this observation suggests that different smell types may have different levels of usefulness in isolating problematic files (e.g., when targeting change-proneness). Definitively confirming this will require further analysis that isolates the effects of different smell types.

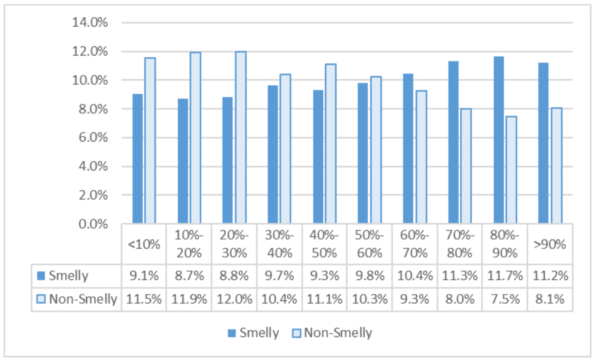
Multiplication factors in Table IV indicate that the issue rates in the smelly files increase 10%–83% over clean files. These values are relatively consistent across the three views for all systems except Hadoop and Wicket. In the cases of Hadoop and Wicket, the multiplication factors under the ARC view are somewhat lower as compared to the other two views. This observation may be accounted for by the above-discussed effect of different smell types on change-proneness. Further analysis will be required to confirm this.

C. Discussion

The results of hypotheses *H1* and *H2* confirm that smells can manifest themselves in different ways. In virtually all cases we observed, the appearance of smells increases the rates of (1) issues that developers experience and (2) changes required in the system.

One aspect of our empirical results that needs to be considered further is the percentage of smelly files in a given system. If the list of identified smells is contained in a large number of files, showing this information to developers does not provide much benefit. Figure 5 shows an example of the percentage of smelly files in Apache Camel under the ACDC view. This figure is representative of the trends we observed in other subject systems and architectural views. The columns and the left y-axis of the figure show the size of the system, in terms of the number of files, across the analyzed 78 versions of Camel. The lower, shaded portions of the columns show the numbers of smelly files in each version, indicating that significant signs of architectural decay were present starting with the initial versions of Camel. The figure shows that the actual numbers of smelly files tend to increase slowly, compared to the growths rate of the whole system. On average, about 18% of the files in Camel are affected by architectural decay. This is not a prohibitively large portion of a system, especially since engineers can narrow down the list of smelly files by looking for a specific smell or smell category.

We also analyzed the sizes of the smelly files we identified. Previous studies have shown that file size correlates with error rates and churn (e.g., [50]). Thus, we investigated whether the identified *architectural* smells are more likely to appear in large files. To this end, we used git-log to collect the sizes of the smelly files. As a representative example, Figure 6 shows the size distribution of smelly and non-smelly files in Camel. Similar distributions were observed in other systems and under different architectural views. The x-axis is the range of file sizes: >90% indicates the largest 10% of all files in the system, 80%-90% indicates files in the second largest group, etc. The y-axis indicates the percentage of smelly and non-



	<10%	10%-20%	20%-30%	30%-40%	40%-50%	50%-60%	60%-70%	70%-80%	80%-90%	>90%
■ Smelly	9.1%	8.7%	8.8%	9.7%	9.3%	9.8%	10.4%	11.3%	11.7%	11.2%
□ Non-Smelly	11.5%	11.9%	12.0%	10.4%	11.1%	10.3%	9.3%	8.0%	7.5%	8.1%

FIG. 6: SIZE DISTRIBUTION OF SMELLY/NON-SMELLY FILES IN CAMEL (ACDC VIEW). FILE-SIZE RANGES ARE ALONG THE BOTTOM; %S OF IDENTIFIED FILES ARE ON THE LEFT.

smelly files that belong to each size range. For example, 11.2% of the files in identified smells are in the largest 10% (>90%), 11.7% of the files are in the next 10% (80%-90%), etc. Figure 6 shows that the architectural smells affect files in all size ranges. Furthermore, the distribution of smells among the files is relatively consistent and, to a large extent, independent of the file size. For example, in the case of Camel, the largest 50% of files are affected by 54.4% of all identified smells. A similar observation is can be made for non-smelly files. Given the correlation between smells on the one hand and issues (Section V-A) and commits (Section V-B) on the other, this result seems to be at odds with previous results of *code-level* studies: symptoms of *architectural* decay are independent of the sizes of implementation files in which they emerge.

VI. Threats to Validity

The key threats to **external validity** include our subject systems. Although the number of systems we used is large compared to other work on architectural decay, it is still limited. Furthermore, all systems are from Apache and implemented in Java. To minimize the effect of this threat, we selected the systems so that they vary along multiple dimensions, including application domain, number of versions, size, and time frame. Furthermore, all the recovery techniques and smell definitions in this paper are language-independent.

Our study's **construct validity** is threatened by (1) the accuracy of recovered architectural views, (2) the detection of architectural smells, and (3) the relevance of implementation issues. To mitigate the first threat, we applied three architecture recovery techniques (ACDC, ARC, and PKG) that showed the greatest usefulness in an extensive comparative analysis of available techniques [20] and in a study of architectural change during system evolution [27], [8], [44], [45]. The three techniques were developed independently and use different strategies for recovering an architecture. To mitigate the second threat, we selected architectural smell types that have been previously studied on a smaller scale [32], [36], [30], [23], [22], and have been shown to be strong indicators of architectural problems. We further ensured the accuracy of our detection algorithms by adopting the well-knows threshold-setting approach. Finally, to mitigate the third threat, we only collected "resolved" and "closed" issues, i.e., issues that have been verified by developers.

The primary threat to our study's **internal validity** and **conclusion validity** involves the relationship between reported implementation issues and architectural smells. In most cases, our results are statistically significant. However, as reported in Section V-B, we encountered one exceptional case. This motivates the need for a follow-on, finer-grain study of individual smell types under individual architectural views, as well as using other statistical tests to confirm the causality.

VII. Related Work

There have been two major approaches in the studies that have attempted to investigate architectural evolution and decay: the *indirect approach* relies on code-level anomalies, while the *direct approach* uses recovered architectures. Multiple previous studies [15], [31], [37] have tried to use code-level anomalies as a means of detecting higher-level decay. Rather than targeting code smells individually, these studies have tried to identify *groups* of code smells that can point to design-level decay. However, the results of such approaches have limited applicability to the problem we are studying as they do not attempt to identify architectural abstractions or decay therein.

The *direct approach* is based on architecture recovery techniques. Brunet et al. [10] studied the evolution of architectural violations in 76 versions selected from four subject systems by using the reflexion method, a technique for comparing intended and recovered architectures of a system. Rosik et al. [42] conducted a case study also using the reflexion method to assess whether architectural drift, i.e., unintended design decisions, occurred in their subject system and whether instances of drift remain unsolved. Le et al. [29] also used a recovery technique to validate the consistency between implementation and variability model of a software system. Recently, Fontana et al. [16] developed an open-source tool to analyze dependency issues at the architectural level. All of these studies were conducted at a significantly smaller scope than our work. The accuracy of their employed recovery techniques is also unclear. Finally, none of these studies relied on architectural smells as concrete instances of architectural decay. Our study has explicitly targeted these issues of scope (via the number of studied systems and versions), accuracy (by using multiple state-of-the-art recovery techniques that were evaluated independently), and actually studied phenomena (by relying on existing definitions of architectural smells).

VIII. Conclusion

In this paper, we have described an empirical study aimed at providing initial answers to the long-standing research question of how architectural smells manifest themselves in a system's implementation. We presented the largest empirical study to date of architectural decay and its impact in long-lived software systems. We have analyzed several hundred versions of 8 well-known systems, totaling 376 MSLOC. To each system version, we have applied 3 different architecture-recovery techniques and analyzed the recovered architectures for 6 different types of smells. On average, we detected nearly 130 architectural smells per system version in each of the three architectural views. Lastly, we have examined the relationships between collected smells and about 42,000 issues that were extracted from the issue repositories of the subject systems.

Our study has not only highlighted the visible manifestations of architectural decay, but also empirically confirmed an assertion that had previously been discussed prominently in the literature: architectural decay and its symptoms—architectural smells—are undesirable, and they can cause significant problems for a software system. Our empirical results have shown that *implementation* files involved in "smelly" parts of the system's *architecture* are statistically significantly more issue-prone and change-prone than the "clean" files.

The analysis we conducted in this paper can be considered introductory, in that it does not try to draw a fine-grained distinction among individual architectural smells. Nonetheless, the results of our study are useful and promising. The established impact of architectural decay, which reveals itself in the form of issue-proneness and change-proneness, lays the foundation for exploring a range of further research questions. In the short term, we want to consider the volume of change (i.e., SLOC) in our analysis. Our long-term goal is to leverage ARCADE to predict architectural decay and potential future issues.

References

[1] Apache jira. https://issues.apache.org/jira, 2018.

[2] arcade:start [USC SoftArch Wiki]. http://softarch.usc.edu/wiki/doku.php?id=arcade:start, 2018.

[3] Bug prediction at google. http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html, 2018.

[4] git-log. http://git-scm.com/docs/git-log, 2018.

[5] What is an issue. https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html, 2018.

[6] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[7] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE TSE*, 2005.

[8] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, 2016.

[9] I. Bowman, R. Holt, and N. Brewster. Linux as a case study: its extracted software architecture. In *ICSE*, 1999.

[10] J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo. On the evolutionary nature of architectural violations. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012.

[11] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John wiley & sons, 2007.

[12] E. W. Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[13] E. W. Dijkstra. On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer, 1982.

[14] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. 1998.

[15] F. A. Fontana, V. Ferme, and M. Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *Proceedings of the Second International Workshop on Software Architecture and Metrics*, SAM '15, pages 1–7, Piscataway, NJ, USA, 2015. IEEE Press.

[16] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[18] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley Longman Ltd., Essex, UK, 1997.

[19] S. Ganesh, T. Sharma, and G. Suryanarayana. Towards a principle-based classification of structural design smells. *Journal of Object Technology*.

[20] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), IEEE/ACM 28th International Conference on*, 2013.

[21] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. *ICSE*, 2013.

[22] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, 2009.

[23] J. Garcia, D. Popescu, G. Edwards, and M. Nenad. Identifying Architectural Bad Smells. In *13th European Conference on Software Maintenance and Reengineering*, 2009.

[24] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *ASE*, 2011.

[25] S. Hassaine, Y. Guéhéneuc, S. Hamel, and G. Antoniol. Advise: Architectural decay in software evolution. In *Software Maintenance and Reengineering (CSMR), 16th European Conference on*. IEEE, 2012.

[26] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[27] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Proc. Mining Software Repositories, 2015*.

[28] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. Relating architectural decay and sustainability of software systems. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016.

[29] D. M. Le, H. Lee, K. C. Kang, and L. Keun. Validating consistency between a feature model and its implementation. In J. Favaro and M. Morisio, editors, *Safe and Secure Software Reuse*, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] D. M. Le and N. Medvidovic. Architectural-based speculative analysis to predict bugs in a software system. In *Proceeding ICSE '16 Proceedings of the 38th International Conference on Software Engineering*.

[31] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 277–286, March 2012.

[32] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012.

[33] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th ICSE*, ICSE '03, 2003.

[34] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE TSE*, Jan. 2004.

[35] D. L. Metayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), 1998.

[36] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 39–46, 2013.

[37] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[38] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. v. Staa. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *Software Engineering (SBES), 2014 Brazilian Symposium on*, pages 91–100, Sept 2014.

[39] R. L. Ott and M. T. Longnecker. *An introduction to statistical methods and data analysis*. Cengage Learning, 2008.

[40] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[41] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SEN*, 1992.

[42] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly. Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 2011.

[43] R. S. Sangwan, P. Vercellone-Smith, and C. J. Neill. Use of a multidimensional approach to study the evolution of software complexity. *Innovations in Systems and Software Engineering*, 2010.

[44] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[45] A. Shahbazian, D. Nam, and N. Medvidovic. Toward predicting architectural significance of implementation issues. In *IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018.

[46] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 1995.

[47] J. W. Tukey. Exploratory data analysis. 1977.

[48] V. Tzerpos and R. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE)*, 2000.

[49] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[50] H. Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.