Toward a Classification Framework for Software Architectural Smells

Duc Le§ Daniel Link§ Arman Shahbazian§ Yixue Zhao§ Chris Mattmann§‡ Nenad Medvidovic§
§Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{ducmle,dlink,armansha,yixuezhao,mattmann,meno}@usc.edu
‡Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
chris.a.mattmann@jpl.nasa.gov

Abstract—Architectural decay is a commonly occurring phenomenon in long-lived software systems. During a system’s lifetime, its architecture gradually degrades via the appearance of architectural “bad smells”, which are instances of poorly thought-through design decisions. To date, there has been limited in-depth study of this phenomenon, its root causes, its characteristics, or the trends it may follow. Instead, both researchers and practitioners have relied on folklore, and their own intuitions and experiences, when referring to architectural smells and their negative impact on software systems. Two key reasons for the absence of large-scale studies of architectural smells are the lack of a systematic categorization of smells and further the absence of algorithms for automatically detecting smells. In this paper, we propose a framework (1) to classify architectural smells based on their characteristics and (2) to detect the smells based on their symptoms. The categorization includes five different classes of smells, of which we focus on four that can be readily detected with the help of existing architecture recovery and analysis techniques. We illustrate the framework with 11 distinct architectural smells across the four categories. We formally define each smell, present a detection algorithm for it, and illustrate it with an example from real systems.

I. INTRODUCTION

The observation that architectural decay occurs regularly in long-lived systems has been part of software engineering folklore from the beginnings of the study of software architecture [47]. It is widely accepted that, during the lifetime of a system, its architecture changes constantly and decays soon after the system’s inception. Architectural decay is usually caused by unintended addition, removal, and modification of architectural design decisions. Decay makes the actually implemented architectures differ significantly, sometimes fundamentally, from their corresponding designed architectures [51]. Software maintenance tends to dominate the cost and effort in a system’s life cycle Over time, a system’s maintenance is increasingly affected by architectural decay [51], [56]. Once the system’s architecture decays, subsequent modifications become increasingly difficult, costly, and error-prone. At that point, software engineers should ideally restructure the system’s architecture and re-factor its implementation [21], [20], [41] to improve its modifiability. However, if the system’s architectural documentation is missing or obsolete, engineers can only rely on implemented artifacts, and possibly their own (frequently unreliable) knowledge of and experience with the system [26].

To deal with the absence of architectural knowledge, previous studies [15], [34], [44] have tried to prevent architecture degradation by relying on code-level anomalies, especially code smells [16], as means of detecting high-level problems. Code smells manifest themselves in a way similar to architectural decay (e.g., maintenance problems). However, the impact of a code smell is usually very narrow. In fact, an individual code anomaly rarely indicates an architectural problem [45]. Our previous study [45] confirmed this by finding no significant relationship between detected code anomalies and architectural modularity in a system. This conclusion has been borne out by another study [40], which observed that certain architectural issues cannot be characterized by existing notions of code smells or anti-patterns. These findings make sense intuitively: a system may have a sound implementation but deficiencies in its underlying architecture, and vice versa. Solely removing code smells may, in fact, do little to improve the system’s maintainability in the long run. To deal with this limitation, some researchers have tried to identify groups of design-level smells, i.e. code smells that can point to design problems [44]. The results of these attempts have been instructive to our research. However, even in those cases, design smells tend to focus on problems and abstractions (e.g., broken inheritance hierarchy or duplicate class implementations with different APIs) that are not necessarily architectural (i.e., component-related).

A comprehensive study of architectural decay should be founded upon architectural smells, the root cause of architectural decay. Architectural decay can manifest itself in many ways, such as increased numbers and severity of bugs, system efficiency problems, growing maintenance costs, etc. Understanding a system’s architecture and detecting architectural decay has served as a motivation (whether direct or implicit) for a number of software architecture recovery techniques [28], [13], [19], [54], [36], whose shared objective is to analyze a system’s implementation in order to extract its architecture. Despite the prevalence and importance of architectural decay, there has been a dearth of large-scale, empirical studies to investigate its nature. Two key reasons are the lack of a systematic categorization of smells and the absence of algorithms for automatically detecting smells. Our previous work [32], [21], [20] attempted to make some inroads. These
were pilot studies, however; the numbers of smells we studied were small and the smells were not systematically classified. Furthermore, detection algorithms for the identified smells were not provided, hampering direct adoption of our work and rendering its results anecdotal.

This paper aims at improving on the shortcomings of prior research in this area. We provide a systematic framework (1) to classify architectural smells based on their characteristics and (2) to detect these smells based on their symptoms. The classification will provide the foundation to further study the nature of architectural smells, their inter-relationships, and their impact on software systems. The automated detection algorithms will facilitate that study, by allowing the analysis of large numbers of systems and ensuring the repeatability and generality of results.

One goal of this research has been to survey and integrate previously reported architectural smells. For this purpose, we have reviewed major collections of such smells. Garcia et al. \[20\], \[21\] and Mo et al. \[41\] each described 4 smells. We recently gave a short description of 12 different smells \[32\]. After eliminating overlaps across the three lists, 14 different smells remained. From these, we removed two because they are related to software connectors \[20\]. Since none of the software architecture recovery techniques we rely on in this work identify explicit connectors, we are unable to detect connector-based smells automatically, and thus we omit them from this paper. Incorporating explicit connectors into architecture recovery is part of our on-going work. Finally, one additional smell (Ambiguous Interface \[32\]) was removed because its detection would require significant human involvement. Unlike bugs, which can “break” a system and have a visible impact, architectural smells may not manifest themselves in an obvious way, but may cause more subtle (but just as harmful) problems with the system during its lifecycle. A proper study of the impact of architectural smells would require many systems and their versions. Therefore, automated detection algorithms are indispensable at this stage.

Consequently, we settled on 11 architectural smells, all of which violate widely accepted software engineering principles. Each smell defined in the paper falls into one of four categories: (1) **concern-based smells** are caused by inappropriate or inadequate separation of concerns \[11\]; (2) **dependency-based smells** arise due to the system components’ interconnections and interactions; (3) **interface-based smells** are yielded by the deficiency in defining system components’ interfaces; and (4) **coupling-based smells** originate from the couplings of system components. Several smells we have identified are (5) **connector-based smells**, caused by inappropriate definition or use of software connectors. We do not discuss those smells for reasons explained above.

We applied our smell detection algorithms on 409 versions of seven widely used open-source systems, totaling 370 MSLOC. Average analysis times of our algorithms per system version ranged from several seconds to several minutes, demonstrating the algorithms’ practical applicability. On average, our algorithms detected nearly 140 architectural smells per system version, demonstrating the prevalence of this problem. Our on-going work is studying the problem’s impact in practice.

The remainder of the paper is organized as follows. Section II summarizes the related research threads that have been brought together to enable the work described in this paper. Section III describes a formalization of software architectures and our proposed catalog of smells. Sections IV presents our detection algorithms for those smells. Examples of smell instances found in real systems are also provided in this section. Conclusion and future work in the last section (Section V) round out the paper. For the rest of this paper, if there is no specific note, smell will be used to refer to architectural smell.

## II. Foundation

Our work discussed in this paper was enabled by three different research threads: (1) architectural decay detection, (2) definition of architectural smells, and (3) software architecture recovery and its related artifacts from source code.

### A. Architectural Decay Detection

Several studies have attempted to define and investigate the nature of architectural decay. In the literature, there are two major approaches to observing architectural decay: (1) the indirect approach uses code-level anomalies (e.g., code smells) as the means to find architecture-level problems, and (2) the direct approach uses recovery methods to rebuild architectures and subsequently find the problems therein (e.g., via architectural smells or decay metrics).

The indirect approach emerged from the research on code-level anomalies, especially code smells. Previous studies by Fontana et al. \[15\], Macia et al. \[34\], Oizumi et al. \[44\] have tried to detect high-level problems (i.e. architecture degradation) by using code-level anomalies. Rather than focusing on isolated code smells, those researchers have tried to identify groups of code smells that can point to design problems, i.e. design smells.

However, as we discussed in the introduction, the results of this approach are very limited. Furthermore, code-level elements are not expressive enough to capture many architectural issues \[40\].

The direct approach is based on automated software architecture recovery techniques, which have been around for over three decades \[5\], \[27\], \[49\], \[50\]. Two studies have examined architectural decay by using the reflexion method, a technique for comparing intended and recovered architectures. Brunet et al. \[8\] studied the evolution of architectural violations in 76 versions selected from four subject systems. Rosik et al. \[48\] conducted a case study using the reflexion method to assess whether architectural drift occurred in their subject system and whether the instances of drift remain unsolved. Hassaine et al. \[25\] presented a recovery technique, which they used to study decay in six versions of three systems. Gurp et al. \[55\] conducted two case studies of software systems to better understand the nature of architectural decay and how to prevent it.

Although there have been studies on architectural decay using recovery methods to directly point out decay in architectures...
of software systems, they were conducted in small scopes. Furthermore, most of these studies did not rely on architectural smells as concrete instances of architectural decay. These are the motivations for us to propose our categorization of architectural smells and detection algorithms in this paper.

B. Definition of Architectural Smells

Similar to the concept of smells at other levels of system abstraction (namely, code and design smells), architectural smells are ultimately instances of poor design decisions [39] at the architectural level. They have a negative impact on system life cycle properties, such as understandability, testability, extensibility, and re-usability [20]. While code smells [16], anti-patterns [9], or structural design smells [18] originate from implementation constructs (e.g., classes), architectural smells stem from poor use of software architecture-level abstractions – components, connectors, interfaces, patterns, styles, etc. Instances of architectural smells detected in a system’s as-is architecture (as opposed to its idealized, as-designed architecture) are candidates for restructuring [7]. Removing architectural smells from a system helps to prevent architectural decay, which in turn improves the system’s quality.

C. Software Architecture Recovery

The direct approach to detecting architectural smells is based on the assumption that a system’s architecture is recoverable from its implementation. In this subsection, we will overview techniques and tools that help to extract architecture-related information and rebuild architectures from this information.

Collecting Links and Couplings – Links and couplings represent connections among architectural components and implementation entities of a system. Links are the channels over which components transfer data and control via their interfaces. In addition to traditionally considered explicit links, we also considered implicit couplings, which have been shown to play a significant role in detecting architectural problems [56]. Couplings are entities in a code base that are required to be updated at the same time, even though there may be no explicit links between those parts. Couplings therefore tie architectural components to each other during the evolution of the system. There are two types of couplings: Co-change (Co) and Duplicate (Du). Co-changes represent coupled components that tend to be modified together during a system’s lifetime. Duplicates (a.k.a. clones) represent coupled components with identical pieces of code. Different tools exist to collect information of links and couplings. For example, Dependency Finder [3] collects links, Code Maat [2] collects co-changes, and PMD [11] collects duplicates. The output of those tools consists of lists of links and couplings among implementation-level entities. In turn, they are used as inputs to architecture recovery techniques, described below.

Collecting Concerns – While links and couplings provide structural information about a software system, concerns represent roles, responsibilities, concepts, or purposes of the system’s entities [22]. Prior work [37]. [4] has shown the usefulness of natural language processing (NLP) techniques for understanding software architecture. Our own prior work has resulted in ARC, a concern-based recovery technique whose accuracy was demonstrated empirically [19]. [31]. Concerns can be extracted from the documentation of a system, comments in its source code, or the source code itself. A common technique for extracting concerns is Latent Dirichlet Allocation (LDA) [6]. Our work has used MALLET [38], an NLP tool with topic modeling capabilities, to extract concerns from source files. This information serves as the input to architecture recovery.

The ARCADE Workbench – Architecture recovery is the process of extracting a system’s architectural information (e.g., its component structure) from lower-level artifacts (e.g., source code) [23]. Depending on the criteria employed by a recovery technique, entities are clustered into components differently, forming different architectural views [29]. To facilitate the study of software architectures of existing systems, we have developed ARCADE, a workbench that employs a set of tools for software architecture recovery and analysis [31]. ARCADE currently provides access to ten recovery techniques. It allows an engineer (1) to extract multiple architectural views from a system’s codebase and (2) to study the architectural changes during the system’s evolution as reflected in those views. This work put us in a good position to select recovery techniques for the study of architectural smells. We have leveraged ARCADE to perform extensive comparative analyses of state-of-the-art recovery techniques [19]. [31]. Furthermore, we have performed pilot studies of architectural smells in the context of several subject systems [35]. [41]. This work suggests ACDC [53], ARC [22] and PKG [31] as a practical cross-section of the ten recovery techniques that (1) exhibit good accuracy and scalability, and (2) approach recovery from different, complementary angles: ACDC leverages a system’s entity dependencies; ARC relies on retrieved concerns; and PKG reflects the system’s implementation organization.

III. Classifying Architectural Smells

This section, first, provides a formalization of architectural concepts and, then, defines architectural smells using those concepts. The formalization also helps us define smell detection algorithms in Section IV. Such a formal definition allows us to precisely delimit similarities and differences among different smells, smell categories, and algorithms. Our previous work [41] already provided a preliminary formalization of architectural concepts to enable the definition of architectural smells. However, that formalization was driven by the specific examples we considered and lacked several elements needed to express the architectural smell categories considered in this paper. Therefore, we revisited and extended it in this paper.

Our definitions of architectural concepts are not intended to cover all aspects of software architecture. However, we would like to ensure that our formalization is expressive enough to define a set of architectural smells that are important and representative. We carefully considered the literature to find aspects of software architecture that are valuable to architects and engineers. These include dependencies and interfaces of components as fundamental architectural building blocks [47].
duplications and logical couplings as root causes of architectural debt [56], concerns extracted from code to help engineers understand the responsibilities of components [4], [37], and so on. Software connectors (and connector-based smells [20]) are another important architectural aspect. However, none of the existing recovery techniques are able to reliably differentiate between components and connectors. For this reason, we have to exclude connector-based smells from our catalog for the time being. Addressing them properly requires significant additional research to modify existing recovery techniques or come up with new ones, which is outside the scope of this paper.

As discussed above, we have reviewed the existing collections of architectural smells [20], [41], [21], [32], and have leveraged the foundational work on architectural patterns and styles, as well code- and design-level smells. Our selection of smells was also guided by widely accepted software engineering principles (e.g., separation of concerns [11], principle of modularity [46]). Furthermore, in order to be practically usable, especially on large, multi-version systems, the smell detection algorithms need to be able to execute automatically or with minimal human assistance. Our resulting catalog contains 11 different architectural smells that satisfy these characteristics and cover four important aspects of software architecture: concerns, dependencies, interfaces, and couplings. Our categorization has similarities with well-established categorizations of smells at the code level [41], [23], in that they both include inter-element (explicit and implicit interconnections) as well as intra-element (responsibilities and exposed interfaces) smell types. However, our focus on architectural abstractions induces some fundamental differences, rendering this work novel.

A. Formalization of Architectural Concepts

Figure 1 shows a notional software architecture $A$ that comprises two components, $C_1$ and $C_2$. Each component contains multiple implementation-level entities. Between entities, links are presented by solid arrows and couplings by dashed lines.

A software system’s architecture is a graph $A$ whose vertices represent the system’s set of components $C$, and whose topology represents the connections embodied in the set of links $L$ and the set of couplings $Cp$ between these components.

$$A = (C, L, Cp)$$

For our purposes, a system is a tuple that consists of architecture $A$ and a nonempty set of topics (i.e., concerns) $T$ addressed by the system. Each topic is defined 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.

$$S = (A, T)$$

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

A component can be either simple or composite. A composite component is an architecture in its own right, allowing for multiple levels of architectural abstraction. Its definition is therefore the same as that for architecture $A$ above. Our definition of architectural smells does not depend on architectural hierarchy, and we thus exclude composite components from our formalization. Additionally, since automated architecture-recovery techniques overwhelmingly focus on components, we do not distinguish between components and connectors here.

A component is a tuple comprising the component’s internal entities $E$ and the probability distribution $\theta$ 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 elements $ie$ that expose that component’s functionality or data), a set of links $L_E$, and a set of couplings $Cp_E$ to other entities. In the object-oriented paradigm, entities are classes and interfaces are public methods.

$$C = \{c_i \mid i \in \mathbb{N}\} \quad c = (E, \theta_c)$$

$$E = \{e_i \mid i \in \mathbb{N}\} \quad e_i = (I_i, L_{E_i}, C_{pE_i}) \quad I = \{ie_i \mid i \in \mathbb{N}\}$$

Both a link $l$ and a coupling $cp$ consist of a source $src$ and a destination $dst$, which are 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 $A$. When composing the two unions, we convert the entities $dst$ and $src$ to their parent components, so that $L$ and $Cp$ are links and couplings between components.

$$L_E = \{l_i \mid i \in \mathbb{N}_0\} \quad l = (src, dst)$$

$$C_{pE} = Co \cap Du \quad C_{pF} = \{cp_i \mid i \in \mathbb{N}_0\} \quad cp = (src, dst)$$

$$L = \bigcap_{i=1}^n L_{E_i} \quad C_p = \bigcap_{i=1}^n C_{pE_i}$$

B. Formalization of Architectural Smells

The architectural smells considered in this paper fall into one of four categories:

1) **concern-based smells** are caused by inappropriate or inadequate separation of concerns;
2) **dependency-based smells** arise due to the system components’ interconnections and interactions;
3) **interface-based smells** are yielded by the deficiency in defining the system components’ interfaces; and
4) **coupling-based smells** appear because of logical couplings across the system components.

We define the individual architectural smells within each category below.

**Concern-based smells** include scattered parasitic functionality and concern overload.
Scattered Parasitic Functionality (SPF) describes a system in which multiple components are responsible for realizing the same high-level concern while some of those components are also responsible for additional, orthogonal concerns. Such an orthogonal concern “infests” a component, akin to a parasite. Orthogonal concerns are either defined by domain experts or NLP techniques (e.g., cosine similarities). Components in an SPF smell violate the principle of modularity \(^{[46]}\). Presence of the smell reduces the understandability and maintainability of the system. For example, if multiple components implement the “networking” concern, fixing a networking problem might require developers to check all these components even though some of the components may primarily focus on other concerns.

Formally, a set of components SPF suffer from this smell iff
\[
\exists z \in T \mid (|SPF| > th_{spf}) \land ((\forall c \in SPF) (P(z \mid c) > th_{spf}))
\]
where \(0 \leq th_{spf} \leq 1\) specifies the acceptable degree of scattering per concern, while \(th_{spf}\) captures that scattering of a topic is allowed to occur across a given number of components before they are considered to be affected by this smell.

Concern Overload (CO) indicates that a component implements an excessive number of concerns. CO violates the principle of separation of concerns \(^{[11]}\). It may increase the size of a component, hurting its maintainability. Formally, a component \(c\) suffers from this smell iff
\[
|z_j| \mid (j \in \mathbb{N}) \land (P(z_j \mid c) > th_{zo})
\]
where \(0 \leq th_{zo} \leq 1\) is the threshold indicating that a topic is significantly represented in the component, while \(th_{zo} \in \mathbb{N}\) is a threshold indicating the maximum acceptable number of concerns per component.

**Dependency-based smells** include dependency cycles and link overloads.

**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. Furthermore, an issue occurring in one component can potentially propagate to all the other components in the cycle. This high coupling between components violates the principle of modularity. Formally, this smell occurs in a set of three or more components iff
\[
\exists l \in L \mid (\forall x \mid (1 \leq x \leq k) \mid (x < k) \implies (l.src \in c_x.E \land l.dst \in c_{x+1}.E)) \land (x = k) \implies (l.src \in c_x.E \land l.dst \in c_1.E)
\]

**Link Overload (LO)** is a dependency-based smell that occurs when a component has interfaces involved in an excessive number of links, affecting the system’s separation of concerns and isolation of changes. For example, modifications to a component with LO due to an excessive number of outgoing links can potentially affect every component to which it is linked. Conversely, any component with an excessive number of incoming links becomes brittle because it is affected by many other components. Formally, a component \(c\) suffers from both incoming and outgoing link overload iff
\[
|\{l \in L \mid l.src \in c.E\}| + |\{l \in De \mid l.dst \in c.E\}| > th_{lo}
\]
where \(th_{lo}\) is a threshold indicating the maximum number of links for a component that is considered reasonable. Excessive incoming links and outgoing links are defined analogously.

**Interface-based smells** include unused interfaces, unused components, sloppy delegation, functionality overload, and Lego syndrome.

**Unused Interface (UI)** is an interface of a system entity that is linked to no other entities. In this case, we say the entity itself is unused. The unused entity might have been added by developers for future use. However, adding entities without any associated use cases violates the principle of incremental development \(^{[17]}\). Having that unused entity adds unnecessary complexity to the component and the software system which, in turn, hinders software maintenance. Formally, a component \(c \in C\) contains an UI smell in entity \(e \in b.E\) iff
\[
(|e.I| \neq 0) \land (\exists l \in e.L \mid l.dst = e)
\]

**Unused Component (UC)** is a component whose internal entities all exhibit the UI smell. UC inherits all of the negative effects of UI. Formally, a component \(c \in C\) is unused iff
\[
\forall e \in c.E \mid isUnusedInterface(e) = true
\]
where \(isUnusedInterface(e)\) is a function that returns true if entity \(e\) is unused.

**Sloppy Delegation (SD)** occurs when a component delegates to other components functionality it could have performed internally. This inappropriate separation of concerns complicates the system’s data- and control-flow which, in turn, slows down the maintenance of that system. 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. Formally, SD occurs between components \(c_1, c_2 \in C\) iff
\[
\exists l \in L \mid l.src = e_1 \in c_1.E \land l.dst = e_2 \in c_2.E
\]
where \(outLink(e_2) = 0\) \land \(inLink(e_2) < th_{sd}\) \land \(c_1 \neq c_2\)

**Functionality Overload (FO)** occurs when a component performs an excessive amount of functionality. Excessive functionality is another form of inappropriate modularity in a system, and it violates the principles of separation of concerns and isolation of change. Formally, component \(c \in C\) has FO iff
\[
e_i \in c.E \mid \sum^n_{i=1} e_i.I > th_{fo}\]
where \(th_{fo}\) specifies the threshold for an excessively high number of operations, i.e., amount of functionality.

**Lego Syndrome (LS)** occurs when a component handles an excessively small amount of functionality. This smell points to components that do not represent an appropriate level of abstraction or separation of concerns. Formally, a component \(c \in C\) has LS iff
\[
e_i \in c.E \mid \sum^n_{i=1} e_i.I < th_{ls}\]
where \(th_{ls}\) specifies a threshold for an excessively small number of operations, i.e., amount of functionality.

**Coupling-based smells** include duplicate functionality and co-change curling.

**Duplicate Functionality (DF)** affects a component if the component shares the same functionality with other components. Changing one duplicated instance of the functionality without changing the others may create errors or inconsistency in the
system’s behaviors. DF violates the principle of modularity and increases complexity. Changing one duplicated instance requires changing all other instances for consistency. Formally, a component $c \in C$ has a duplicate functionality iff

$$e_i \in c . E \mid \sum_{i=1}^{n} |e_i . Du| > th_{df},$$

where $th_{df}$ specifies a threshold for an excessively high number of duplications.

**Co-change Coupling (CC)** occurs when changes to an entity of a given component also require changes to an entity in another component (recall Section II). CC has similar negative consequences to DF. Specifically, making a single change to a system affected by CC smells might require engineers to check and make changes to multiple components each time. Formally, a component $c \in C$ has a co-change coupling iff

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

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

### IV. Detecting Architectural Smells

Although architectural smells have been discussed in literature, a large-scale empirical study of architectural smells and their impact is still missing. One reason for this is the lack of automated smell detection algorithms: without an efficient way to detect architectural smells from a system’s implementation, it is impossible to scale up an empirical study to many systems with many versions. This state of affairs drives us to define detection algorithms for our proposed smell catalog.

This section describes (1) three smell detection strategies and (2) corresponding detection algorithms for the 11 architectural smells defined in Section III. A detection strategy is a general approach to detect a group of smells. We defined four categories of smells, and smells in each category are detected based on different detection strategies or combinations of strategies.

Taking advantage of prior research on specifying and detecting smells at lower abstraction levels (i.e., code and design smells), we made use of the resulting detection strategies. Three popular smell detection approaches are lexicon, structure, and measurement. We applied the detection strategies that are related to the characteristics of each smell category. Furthermore, we also selected some of the best-proven practices (e.g., how to select metric thresholds) from the literature.

All architectural-smell detection algorithms have been implemented and integrated into the ARCADE workbench (recall Section II). We have applied ARCADE to several systems, to detect smells in their architectures. Each system has a large number of versions, allowing us to study the smells over the evolution of the system.

Specifically, to study the effectiveness, performance, and scalability of our algorithms, we applied them on a total of 409 versions of seven widely used Apache open-source systems: Camel, Continuum, CXF, Hadoop, Nutch, Struts2, and Wicket. The total amount of code analyzed by our algorithms was 370 MSLOC. Different systems’ individual versions ranged in size from 118 KSLOC (Nutch) to 1.96 MSLOC (Hadoop). Average analysis times of our algorithms per system version ranged from a few seconds (for `detectUI_UC` shown in Algorithm 4 below) to 10 minutes (for `detectSD` shown in Algorithm 5). The total number of smells detected for all 1,227 architectural models across the seven systems (i.e., three views—ACDC, ARC, and PKG—per system version) was 169,667. This averaged to 138 architectural smells per individual system version under a particular architectural view. Our preliminary analysis has also identified that different smells tend to appear with different frequencies, and that those frequencies depend on the system. However, a deeper analysis of such trends and the causes behind them is beyond the scope of this paper; it is part of our on-going work.

For illustration purposes in this section, given the space constraints, we will only highlight one instance of each smell per system. To that end, we selected two Apache systems that contain examples of all 11 smells defined above: CXF, a widely used open source web services framework, and Nutch, a large open source web crawler, the parent to Hadoop. Table I shows the two subject systems, the respective numbers of versions we analyzed, and the average sizes of each version.

**TABLE I**

**SUBJECT SYSTEMS ANALYZED IN OUR STUDY**

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>No. Versions</th>
<th>Avg. SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache CXF</td>
<td>Service Framework</td>
<td>120</td>
<td>915K</td>
</tr>
<tr>
<td>Apache Nutch</td>
<td>Web Crawler</td>
<td>21</td>
<td>118K</td>
</tr>
</tbody>
</table>

**A. Smell Detection Strategies**

1) **Lexicon:** The lexicon-based detection strategy relies on NLP techniques. This strategy is appropriate for detecting concern-based smells. As we mentioned in the previous section, we use the LDA implementation in MALLET, a topic modeling tool. For each system, we build a topic model, which contains a list of topics and their associated keywords, from the textual contents of its source files. We refer to those topics as the concerns of the system. We then use the extracted topic model to compute the concern distribution of each entity in the system. The resulting list of concern distributions can be used to detect violations of the separation of concerns in architectural components (i.e., concern-based smells).

2) **Structure:** The structure-based detection strategy relies on different types of interconnections among components. In our approach, if two entities are connected by a link or a coupling, we say their parent components are connected by that link or coupling. Dependency-based and coupling-based smells are undesirable structural patterns formed by interconnections (links and couplings) among architectural components. Therefore, this strategy is appropriate for detecting smells in those two categories.

3) **Measurement:** The measurement-based detection strategy relies on the values of different software metrics. One can reuse classic software metrics suites, such as CK and MOOD, or define new metrics to this end. This approach has also been used extensively to detect code smells. This strategy can be used by itself (e.g., to detect interface-based...
smells, such as *Unused Interface* or as a supplement to other strategies (e.g. combine with lexicon-based strategy to detect Concern Overload).

One critical issue in the measurement strategy is where to set thresholds, i.e., defining the criteria that classify a given value of a metric to be an indicator of a smell. In our detection algorithms, we set thresholds by using InterQuartile analysis [52], which is an efficient technique for detecting outliers (e.g., 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 inter-quartile range \((iqr)\) is defined as the interval between \(q_1\) and \(q_3\), \(q_3 - (1.5 \times iqr)\) and \(q_3 + (1.5 \times iqr)\) are defined as inner fences, that mark off the “reasonable” values from the outliers. “Inner fences” have been used widely in research on software metrics [14] as thresholds to find outliers. In this section, we will use two shorthand functions \(\text{getLowThreshold()}\) and \(\text{getHighThreshold()}\), which accept a list of values and return the low and high values of the “inner fences”.

### B. Smell Detection Algorithms

**Algorithm 1: detectSPF**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C): a set of components</td>
<td>(T): a set of system concerns</td>
</tr>
</tbody>
</table>

*SPFxsmells*: a map where keys are concerns and values are components

1. \(SPFxsmells \leftarrow \text{initialize map as empty}\)
2. \(\text{concernCounts} \leftarrow \text{initialize all concern counts to 0}\)
3. For \(c \in C\) do
   4. \(T_c \leftarrow \text{getConcernsOfComponent}(c)\)
   5. \(\text{th}_c \leftarrow \text{getHighThreshold}(P(T_c))\)
   6. For \(z \in T_c\) do
      7. If \(P(z | c) > \text{th}_c\) then
         8. \(\text{concernCounts}[z] \leftarrow \text{concernCounts}[z] + 1\)
   9. \(\text{th}_\text{spf} \leftarrow \text{getHighThreshold}(\text{concernCounts})\)
10. For \(z \in T\) do
    11. If \(\text{concernCounts}[z] > \text{th}_\text{spf}\) then
        12. For \(c \in C\) do
            13. If \(P(z | c) > \text{th}_c\) then
                14. \(\text{SPFxsmells}[z] \leftarrow \text{SPFxsmells}[z] \cup \{c\}\)

**Concern-based smells**

*Scattered Parasitic Functionality (SPF)*: Algorithm [1] detectSPF, returns a map \(\text{SPFxsmells}\) where each key in the map is a scattered concern \(z\), and each value is a set of at least \(\text{th}_\text{spf}\) number of components that have the corresponding key \(z\) above the threshold \(\text{th}_z\). Lines [10][14] create a map where keys are concerns and values are the number of components that have that concern above threshold \(\text{th}_z\). The \(\text{getConcernsOfComponent}\) function in Line [4] returns the topic distribution of the component, which is computed by MALLET. Line [3] calculates the threshold \(\text{th}_z\) for each component \(c\), which helps determine representative topics of a component (Line [7]). Line [2] calculates the threshold \(\text{th}_\text{spf}\) for the maximum number of concerns in a component. Both thresholds are determined dynamically by the InterQuartile method. Lines [10][14] identify SPF instances by checking if concern \(z\) appears in at least \(\text{th}_\text{spf}\) components (Line [13]).

In most versions of CXF, under the ARC view, we found a Scattered Parasitic Functionality instance where the *org.apache.cxf.BusFactory* entity and its subclasses are scattered across different components. Even their parent packages are different. Although those classes address the same big concern, which is “bus”, the subclasses implement different specific concerns. This is the main reason for ARC to assign them to different components.

*Concern Overload (CO)*: Algorithm [2] detectCO, determines which components in the system have CO. The algorithm operates in a manner similar to detectSPF. detectCO begins by creating a map, \(\text{componentConcernCounts}\), where keys are components and values are the number of relevant concerns in the component (Lines [5][9]). While creating the map, threshold \(\text{th}_z\) is dynamically computed for each component (Line [5]) and used to determine prevalent concerns in each component. Later, detectCO uses that map to compute threshold \(\text{th}_\text{co}\) in Line [9] which is then used to determine which components have the CO smell (Lines [10][12]).

Like Scattered Parasitic Functionality, we also found a long-lived Concern Overload instance under the ARC view. We found a component that contains the most classes in the *org.apache.cxf.phase* package. This component implements different steps of information processing in CXF, which include reading a message, transforming it, processing headers and validating the message. Although all these steps are related to message handling, putting all of them into a single component causes it to have the CO smell.

**Algorithm 2: detectCO**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C): a set of components</td>
<td>(T): a set of system concerns</td>
</tr>
</tbody>
</table>

*COsmells*: a set of Concern Overload instances

1. \(\text{COsmells} \leftarrow 0\)
2. \(\text{componentConcernCounts} \leftarrow \text{initialize all brick concern counts to 0}\)
3. For \(c \in C\) do
   4. \(T_c \leftarrow \text{getConcernsOfComponent}(c)\)
   5. \(\text{th}_c \leftarrow \text{getHighThreshold}(P(T_c))\)
   6. For \(z \in T_c\) do
      7. If \(P(z | c) > \text{th}_c\) then
         8. \([\text{componentConcernCounts}[z]] = [\text{componentConcernCounts}[z]] + 1\)
   9. \(\text{th}_\text{co} \leftarrow \text{getHighThreshold}(\text{componentConcernCounts})\)
10. For \(c \in C\) do
    11. If \([\text{componentConcernCounts}[c]] > \text{th}_\text{co}\) then
        12. \(\text{COsmells} \leftarrow \text{COsmells} \cup \{c\}\)

**Dependency-based smells**

*Dependency Cycle (DC)*: We detect DC smells by identifying strongly connected components in a software system’s architectural graph \(G = (C, L)\). A strongly connected component is a graph or subgraph where each vertex is reachable from every other vertex. Each strongly connected component in \(G\) is a Dependency Cycle. Any algorithm that detects strongly
connected components [12, 33] can then be used to identify DC. Therefore, we do not include the detection algorithm for DC in this paper.

Both Nutch and CXF have one instance of Dependency smell starting with their early versions. These instances are persistent throughout both systems’ life cycles. As both system evolve, the cycles increase in size by involving more components. This observation holds with all three architectural views.

**Link Overload (LO):** Algorithm 3 detectLO, extracts the LO variants for a set of components C by examining their links L. The algorithm first determines the number of incoming, outgoing, and combined links per component (Lines 4-6). detectLO sets the threshold \( t_{th} \) for each variant of LO by computing the thresholds \( t_{th} \) for incoming, outgoing, and combined links (Lines 7-8). The last part of detectLO identifies each component and the directionality that indicates the variant of LO the component suffers from (Lines 9-12).

In Nutch, we found that some components, especially those which are related to web user interfaces, suffered from Link Overload, such as org.apache.nutch.webui.pages.crawls or org.apache.nutch.webui.pages.instances in the PKG view. These components were created with many inner classes within the main classes.

Algorithm 3: detectLO

```java
Input: C: a set of components, L: links between components
Output: LOsmells: a set of Link Overload instances
1 LOsmells ← ∅
2 numLinks ← initialize map as empty
3 directionality ← (“in”, “out”, “both”)
4 for c ∈ C do
5    for d ∈ directionality do
6        numLinks[<c, d>] ← numLinks[<c, d>] + getNumLinks(c, d, L)
7    for d ∈ directionality do
8        t_{th}[d] ← getHighThreshold(numLinks, d, C)
9    for c ∈ C do
10       for d ∈ directionality do
11          if getNumLinks(c, d, L) > t_{th}[d] then
12             LOsmells ← LOsmells ∪ {{<c, d>}}
```

**Interface-based smells**

**Unused Interface (UI) and Unused Component (UC):** As we mentioned in Section III-B, UC is the extreme case of UI. Algorithm 4 detectUI_UC, allows us to detect both of these smells. detectUI_UC uses the set of links L to determine if an interface has been used or not. The algorithm checks every entity in each component (Lines 2-7), and if an entity has a public interface but no link, then the entity and its parent component are added to the UI instances list. Line 2 uses a boolean flag, isUB, to mark that an component does not have UC if at least one entity in the component does not have UI. Line 11 checks and adds UC instances to the smell list.

In Nutch, we found that the SequenceFileInputFormat class was unused in some 1.x versions (in each view, the parent component of this class was affected by the Unused Interface smell). This class had been removed in version 2.0. We assume that developers noticed and decided to remove this unused class.

We could not find any instances of Unused Component in Nutch, but in CXF. Under the PKG view, the org.apache.cxf.simple component had been unused from version 2.0.6 to version 2.2.9. Later, it was used again from version 2.2.10.

Algorithm 4: detectUI_UC

```java
Input: C: a set of bricks, L: links between components
Output: UIsmells: a set of Unused Interface instances, UCsmells: a set of Unused Brick instances
1 UIsmells ← ∅, UCsmells ← ∅
2 for c ∈ C do
3    isUC ← true
4    for e ∈ c.E do
5       if getNumInterfaces(e.L) > 0 then
6          if getNumLinks(e.L) = 0 then
7             UIsmells ← UIsmells ∪ {{<c, e>}}
8          else
9             isUC ← false
10     if isUC then
11        UCsmells ← UCsmells ∪ {{<c>}}
```

**Sloppy Delegation (SD):** Algorithm 5 detectSD requires a threshold \( t_{sd} \), which defines the minimum number of in-links to consider a delegation appropriate. The algorithm checks every link in each entity (Lines 2-5), and if a link has a ‘dst’ entity which satisfies the checking condition of SD (defined in Section III-B), then the ‘dst’ entity and its parent component are added to the list of SD instances (Line 10).

In Nutch, we found that org.apache.nutch.crawl and org.apache.nutch.scoring.webgraph are two components which are heavily affected by the Sloppy Delegation smell. Out of the three architectural views, ACDC provides a better clustering to reduce the number of SD smells. However, those two components still need refactoring.

Algorithm 5: detectSD

```java
Input: C: a set of bricks, L: links between components, \( t_{sd} \): threshold for relevance delegation
Output: smells: a set of Sloppy Delegation instances
1 smells ← ∅
2 for c ∈ C do
3    for e1 ∈ c.E do
4       for l ∈ e1.L do
5          if l.src = e1 then
6             e2 ← l.dst
7             e2 ← getParent(e2)
8             if (e1 \neq e2) ∧ (getOutLink(e2) = 0) ∧ (getInLink(e2) < t_{sd}) then
9                smells ← smells ∪ {{<c1, e1>, {e2, e2}>
```

**Function Overload (FO) and Lego Syndrome (LO):** These two smells come in a pair, which indicates overloaded and underloaded functionality in components. Algorithm 6 detectFO_LS, allows us to detect both smells in one run. The algorithm first creates a map between components and their numbers of interfaces (Lines 3-5). This map then is used to...
compute two thresholds $th_{fo}$ and $th_{ls}$, i.e., the high (Line 6) and low values (Line 7) of the inner fences, respectively. Finally, the algorithm revisits each component, checking and adding detected smell instances into two smell lists (Lines 8-12).

Across all 1.x versions of Nutch under the PKG view, we found that one component, `org.apache.nutch.crawl`, has Function Overload and two other components, `org.apache.nutch.net.protocols` and `org.apache.nutch.tool.arcc`, have Logo Syndrome.

Under the PKG view of CXF, we found that the component `org.apache.cxf.interceptor` has the Co-changes Coupling smell along with 6 other components in version 2.x. Later on, one CC instance has been removed in version 3.x. However, new CC smell instances were also introduced. Under the PKG view, we also found `org.apache.cxf.ws.policy.validators` and `org.apache.cxf.jaxb` to be strongly affected by Duplicate Functionality. Almost all entities in those two components have duplications with entities in other components.

V. CONCLUSION AND FUTURE WORK

This paper presented a classification framework for software architectural smells. The framework is intended to serve as a systematic reference that helps engineers to relate one smell to another by considering the architectural elements they affect. We have classified and formally defined 11 smells across four categories, and provided algorithms that allow engineers to automatically detect these smells from a system’s implementation using existing architectural recovery techniques and software analysis tools. These algorithms have been successfully applied on a large, and growing, corpus of real systems. We have also provided concrete examples of smell instances found in actual systems.

This work constitutes a foundation for further studies of architectural decay. As the primary motivation of this paper, we have been designing an empirical study on how architectural smells manifest themselves in a system implementations. We expect to find the visible manifestations of architectural smells and empirically confirms a finding that had previously only been suspected: the parts of a system that exhibit signs of architectural decay also exhibit more concrete implementation-level problems (e.g., bugs) than the “clean” parts. This will serve as a clear demonstration that engineers should spend time and effort to remedy architectural decay in their systems, a task that is currently often discarded as unnecessary and costly. The empirical study will also help us to analyze the complexity of our detection algorithms in order to optimize them. Furthermore, we would like to investigate the impact of smells individually and in tandem, in order to provide better guidance for engineers to prioritize their maintenance tasks. Our long-term goal is to predict architectural decay and potential future issues based on the available implementation-level information.

REFERENCES
