An Architecture Smell Knowledge Base for Managing Architecture Technical Debt

Paula Rachow
paula.rachow@uni-hamburg.de
Universität Hamburg
Hamburg, Germany

Matthias Riebisch
matthias.riebisch@uni-hamburg.de
Universität Hamburg
Hamburg, Germany

ABSTRACT

Many software projects suffer from architecture erosion and architecture technical debt. One challenge is to identify affected parts and prioritize them for refactoring. Architecture smells are indicators of potential architecture technical debt, but architecture smells are ambiguous and their impact is not always clear. To address this, we have built a knowledge base that improves understanding of architecture smells and identifies violated software design principles and affected quality attributes. The design principles help our understanding of what causes architecture smells, while the impaired quality attributes represent the consequences. We conducted a systematic literature review to identify these relations and built an architecture smell ontology. This ontology provides a knowledge base that architects can use to prioritize the smells according to the project's individual quality goals.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software evolution**; **Risk management**; *Software design tradeoffs*; *Software design engineering*.

KEYWORDS

architecture smell, maintainability, knowledge base, software design principles, ontology, prioritization, architecture technical debt

ACM Reference Format:
Paula Rachow and Matthias Riebisch. 2022. An Architecture Smell Knowledge Base for Managing Architecture Technical Debt. In *International Conference on Technical Debt (TechDebt '22), May 16–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3524843.3528092

1 INTRODUCTION

Often, software systems must continue to perform over the long term and evolve as requirements change. However, evolvement is only possible with high internal quality. During changes, the software architecture degrades because developers knowingly or unknowingly make wrong or sub-optimal design decisions [49]. These wrong or sub-optimal design decisions are one cause for *technical debt* (TD), particularly *architecture technical debt* (ATD) which focuses explicitly on TD at the architecture level [52]. Delayed repayment of ATD can be more extensive and costly the later it is accomplished [3]. Thus, practitioners must repay this continually accumulated debt frequently for the system to evolve.

One way to restore crucial quality attributes and repay ATD is refactoring. Refactoring requires significant effort and experienced developers, which are both scarce resources. Especially on the architecture level, developers often must coordinate refactoring steps with other developers, as they may have a system-wide impact. Therefore, developers cannot address all ATD by themselves, and ATD must be prioritized as part of the overall planning.

Architecture smells (AS) are indicators of refactoring opportunities by revealing potential ATD. AS can especially indicate unintentional and unknown ATD in the project. However, architecture smells are ambiguous and the impact is not always clear, making them hard to prioritize. Furthermore, the prioritization of AS depends on the domain, business requirements, evolution plans, and team preferences. Hence, prioritization is an individual process for each system.

Therefore, this paper provides the needed base for further research by contributing structured knowledge and classification through an ontology. The ontology helps achieve clarity and common understanding about possible causes and related consequences of AS and, therefore, potentially ATD. We conducted a systematic literature review to identify the relations of AS to maintainability, its sub-characteristics, and software design principles.

Practitioners can use techniques such as the architecture tradeoff analysis method (ATAM) [22] to prioritize their individual quality goals and then use the knowledge base to identify AS that most align with those goals. Several factors may affect the quality goals and, therefore, the prioritization of smells. Modifiability and modularity are essential if feature velocity is vital because they influence the time to implement a change and its propagation. If there are a lot of novices and newcomers in the team, the analyzability of the code is crucial. If, in contrast, the team consists of many long-term employees and few new team members have to become acquainted with the system, analyzability can have a lower impact. Testability may be prioritized in companies where correctness is essential, e.g., in the banking domain.

Through the violated design principles, practitioners can identify and address possible causes of ATD. The design principles help assess if the occurred architecture smell is indeed a problem. Furthermore, because design principles propagate quality attributes, they indicate which quality attribute is impaired.

The paper is structured as follows. Section 2 describes architectural smells, software design principles, quality attributes, and

TechDebt '22, May 16–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9304-1/22/05…$15.00
https://doi.org/10.1145/3524843.3528092

ontologies. Section 3 provides an overview of related work. In Section 4, we present our research questions, outline how we built the ontology, and present the literature review that provided the ontology's input. Section 5 presents the results while Section 6 discusses the results and their implications. Section 7 reports on the threats to the validity of our study, and Section 8 concludes the paper and outlines the directions of future work.

2 BACKGROUND

In this section, we explain the meaning of ATD, AS, software design principles, maintainability, and ontologies in order to avoid confusion or misunderstandings.

2.1 Architecture Technical Debt

Architecture technical debt derives from sub-optimal architectural decisions [32]. There are two types of TD, intentional and unintentional TD [34]. Schmid [46] further distinguishes between potential and effective TD. The first refers to a sub-optimal structure of a system, while the latter additionally has a negative impact on future development.

ATD has to be repaid to let the system evolve, but ATD is hard to detect, and its repayment is wide-ranging and often avoided [52]. As software systems grow, original design choices, e.g., decisions about frameworks or the system's structure, can become constraints, limiting future evolution or even preventing it. This constantly accumulated ATD is not always known which is were AS can be helpful.

2.2 Architecture smells

Architecture smells are indicators for necessary architecture refactorings [48] and, therefore, potential ATD. They could be a sign of violations of architectural rules or just bad practices. Furthermore, they indicate that the architecture is no longer adequate under the current requirements and constraints, which might differ from the original ones [54]. As with code smells, an AS does not always inevitably indicate a problem, but AS point to places in the system's architecture that developers should further analyze [28].

2.3 Software Design Principles

Software design principles are recommendations that engineers should follow during program implementation to achieve a sound system design. They encompass design concepts and promote quality attributes. Robert C. Martin describes the most established ones in his book "Clean Architecture" [31], including the SOLID Principles and component principles such as the common closure principles. Violations of design principles can help understand the causes of AS and, therefore, causes of ATD. They help understand the causes of architecture smells and, therefore, indicate how architecture smells can be resolved.

2.4 Maintainability

We found that maintainability and its sub-characteristics are sometimes used imprecisely or not based on the ISO/IEC 25010:2011 [21], e.g., they are used together with other -ility terms such as understandability and readability. Therefore, we present the definitions from the ISO/IEC 25010:2011 here. In our work, we use the terms characteristic and attribute as synonyms.

Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified. Maintainability is one of the most fundamental aspects of software engineering. Especially as software tends to be long-living and evolving, it is the main factor why software can no longer evolve and becomes noncompetitive.

Maintainability is composed of the following sub-characteristics: **Analyzability** is the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies, causes of failures, or to identify parts to be modified. In other words, analyzability encompasses the effort to diagnose side effects of changes or causes of failures or the effort to determine parts that need to be changed.

Modularity is the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. Poor modularity increases the amount of (unnecessary) changes to neighboring components, i.e., high coupling, low cohesion, or poor separation of concerns decrease modularity.

Modifiability is the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality. Modifiability affects the effort for improvements, bug removal, or adjustments to changes in the environment.

Testability: is the degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component, and tests can be performed to determine whether those criteria have been met. Testability is impaired when components can not be tested separately or the effort for testing is increased, e.g., by tight coupling.

Reusability is the degree to which an asset can be used in more than one system or in building other assets. Reusability is impaired when code is duplicated or a solution is too specific.

2.5 Ontologies

"An ontology is a formal, explicit specification of a shared conceptualization" [20] and defines a vocabulary that describes concepts and relations representative for a domain. It represents a structure of entities to organize knowledge and manage complexity (in a specific domain). There are multiple reasons for developing an ontology. In our case, we want to share a common understanding of information structure among researchers and practitioners, enable reuse of domain knowledge, and make domain assumptions explicit.

An ontology consists of concepts in a domain of discourse named *classes*, *concept properties* describing various features and attributes of the concept, and *restrictions* on properties. An ontology may implement a web of terms, i.e., for a classification or a taxonomy, and together with a set of individual instances of classes constitutes a knowledge base. In reality, there is a fine line where the ontology ends and the knowledge base begins [43].

3 RELATED WORK

We present related work concerning AS catalogs, AS categorizations, and prioritization approaches for AS and ATD. All three topics are important to our work. Catalogs include lists of AS and information needed for our knowledge base. Categories and taxonomies have the advantage over catalogs in that they represent relationships between elements. Lastly, prioritization indicates the impact of AS or similar indicators.

3.1 Architecture Smell Catalogs

There have been some attempts to start a catalog of AS. Lippert and Roock [28] introduced 29 AS and structured them by occurrence, i.e., they differentiated whether they occurred on the dependency graph, inheritance hierarchy, package, subsystem, or architectural layer level. While they provided an overview, they did not interconnect the smells or elaborate on the prioritization.

Brown et al. [6] collected anti-patterns for software architecture. Most of the anti-patterns are focused on the development process and management. Hence they do not conform to the definition of AS. The authors did not include a guideline for prioritizing the smells.

Garcia et al. [18] defined the four AS *Connector Envy*, *Scattered Parasitic Functionality*, *Ambiguous Interface*, and *Extraneous Adjacent Connector* in detail. They added the quality impacts and trade-offs and an example of each smell.

Azadi et al. [4] collected AS that are automatically detectable by a tool. They proposed a categorization following three design principles that the smells violate. They defined the three principles modularity, hierarchy, and healthy dependency structure and classified 19 architecture smells according to these principles.

Mumtaz et al. [40] conducted a systematic mapping study on AS detection. They identified which AS are and which are not detectable by tools and collected a large number of smells. They included a list of all the AS they found, including a definition and categorized them according to detection approaches. They do not discuss them regarding prioritization.

Overall, these catalogs lack interconnection of smells and connection to principles and quality attributes [6, 28], or they report only on a small number of smells [4, 18] and do not cover prioritization [40]. Our knowledge base contains 114 architecture smells and provides information about relations both between AS and maintainability characteristics and between AS and design principles. Both relations can be considered to discover more ATD and evaluate the impact.

3.2 Categorization & Taxonomy

Mantyla et al. [30] defined a taxonomy for code smells, namely *bloaters*, *object-orientation abusers*, *change preventers*, *dispensables*, *encapsulators*, *couplers*, and *others*. These categories are not disjoint; e.g., smells that prevent change can also be couplers. The categories were defined for code smells and are only partly transferable to AS.

Le et al. [25] classified AS by connecting them to metrics and the impacted quality attributes. They did not only use the quality attributes of the ISO standard, but also other terms like complexity and understandability. They defined four categories of smells: interface-based, change-based, concern-based, and dependency-based. The categories help with the understanding, but the authors do not indicate how this categorization helps when addressing the smells.

Simple categories do not seem suitable for AS as they cannot be distinctly assigned or some smells do not fit any categories. We propose an ontology to consider all AS with their interconnections and ambiguities they embody. Our ontology is a more suitable representation for the complex topic of architecture smells.

3.3 Prioritization

De Almeida et al. [11] evaluated their business-driven approach to prioritize technical debt. They considered the perspective of business and technical stakeholders for prioritizing specific TD items of the system according to business values. Furthermore, they identified eight business factors that affect decision-making regarding TD.

Schmid [46] used a mathematical approach to prioritize the system's next evolutionary steps. He estimates the restructuring costs and the probability of evolution steps and distinguishes between potential and effective TD. Potential TD is any sub-optimal software system part, while effective TD refers to issues in the software system that make further development more complex. Hence, according to him, only effective TD should be considered for refactoring.

Lenarduzzi et al. [27] conducted a systematic literature review on technical debt prioritization. They provided an overview of the different TD types and an impact map with factors and measures related to the interest of TD considered when prioritizing.

Arcoverde et al. [2] prioritized code anomalies by calculating the architectural impact. They developed four different heuristics which are individually usable. They calculated the change density (over time), error density (over time), anonamy density (static code analysis), and architecture role (by architects). This work only considers code smells and not architecture smells.

Fontana et al. [14] computed the PageRank and criticality of the three AS *unstable dependency*, *hub-like dependency*, and *cyclic dependency* and suggest using the combination to find the most crucial architecture smells.

Martini et al. [33] performed a case study to research how practitioners prioritize technical debt. They asked for the impact and refactoring cost for the three smells mentioned above. In a later study, Fontana et al. [13] asked practitioners again about the impact, refactoring effort, and priority of eight AS.

The existing prioritization approaches for architecture smells only considered three to eight smells, and they did not include the quality attributes. Our knowledge base includes considerably more smells and can easily be extended to future arising smells. The connections to quality attributes and design principles help to assess the impact and identify effective TD that has to be repayed and, therefore, supports the prioritization of the smells.

4 RESEARCH METHOD

In order to implement our knowledge base, we defined research questions, built an ontology, and conducted a systematic literature review based on the research questions.

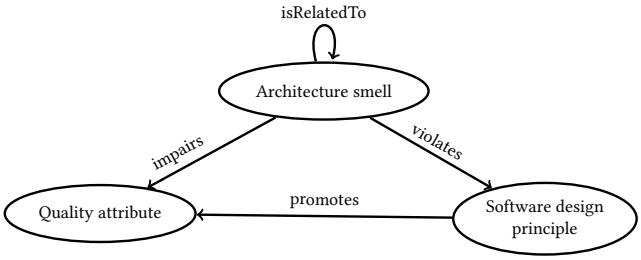

Figure 1: Relations in the ontology

4.1 Research Questions

Our aim is to prioritize ATD by using architecture smells. As they lack clarity, we first need to improve the understanding of AS. To enable this, we considered the following research questions:

RQ1: *Which architecture smell violates which software design principle?*

Many definitions of AS state that they violate commonly known software design principles [4, 16, 23], but they often do not name a specific one. With this research question, we want to identify the specific connections.

RQ2: *Which architecture smell impairs maintainability or rather one of the maintainability sub-characteristics?*

The impact of architecture smells is not always clearly defined, but it is crucial for prioritization. To illustrate the impact more clearly, we want to find explicit connections of AS to quality attributes. Architects can then prioritize them according to the system's quality goals.

4.2 Development of Ontology

We choose an ontology to implement the architecture smell knowledge base for different reasons: The ontology enables to share a common understanding of the structure of information, enables the reuse of knowledge, makes assumptions explicit, and allows to analyze the knowledge [43]. Furthermore, the ontology provides high flexibility and extensibility.

We followed the structured process proposed by Noy and McGuinness [43] to develop our ontology.

Domain and scope of the ontology. The purpose of the ontology is to organize the different architecture smells and connect them to the design principles and quality attributes they impact. Thus, we used the above-defined research questions as competency questions.

To our knowledge, there does not exist an ontology for architecture smells. There have only been attempts to build a catalog (see Section 3.1).

Classes and the class hierarchy. We define the classes *"architecture smell"*, *"software design principle"*, and *"quality attribute"*. They each contain their specific instances. Each subclass of "architecture smell" is a specific smell, e.g., ambiguous interface [25]. Each subclass of "software design principle" is a concrete design principle, e.g., single responsibility principle [31]. The *"quality attribute"* class consists of the *maintainability* subclass, which in turn has its sub-characteristics as subclasses.

Properties. We define the four properties *"promotes"*, *"impairs"*, *"violates"* and *"isRelatedTo"*. Figure 1 shows the connection between the classes and the properties. A software design principle promotes a quality attribute. An architecture smell violates a software design principle and impairs a quality attribute. We use the general name "isRelatedTo" to connect AS that represent a subclass or are close but are not exactly the same smell. Examples are *"cyclic dependency"* [47] and *"unnecessary dependencies"* [48]. Not every *cyclic dependency* has to be unnecessary, but there are instances where a *cyclic dependency* would present itself as a subclass of *unnecessary dependencies*.

Representation. We used the Protégé [41, 42] tool to support the formalization of the ontology in the formal Web Ontology Language (OWL) [39]. With the help of the tool, connections of equivalent classes and subclasses are linked automatically.

4.3 Literature Review

We conducted a systematic literature review to identify the connections between architecture smells, software design principles, maintainability, and its sub-characteristics. Our literature review follows the guidelines established by Kitchenham and Charters [24] and is depicted in Figure 2. We used the results to populate our ontology.

Data sources and search strategy. We used the five well-established digital libraries IEEExplore, ACM Digital Library, SpringerLink, ScienceDirect, and Web of Science. To check if we did not miss any significant literature, we double-checked at least the first fifty results in Google Scholar. We searched these databases and libraries to obtain information related to architectural smells and connected software design principles and quality attributes in January 2022. We formulated two search strings to locate the related work.

To reveal the violated software principles, we chose:

> ("architecture smell" OR "architecture bad smell" OR "architectural bad smell" OR "architectural smell" OR "architectural defect" OR "architecture anti-pattern" OR "architecturally-relevant code smell" OR "hotspot pattern") AND principle

First, we included the term "architecture smell" and known synonyms from the literature [5, 8, 15, 35, 36, 45]. Second, we added the term "principle" to find the connected software principles.

To find the connected quality attributes, we used:

> ("architecture smell" OR "architecture bad smell" OR "architectural bad smell" OR "architectural smell" OR "architectural defect" OR "architecture anti-pattern" OR "architecturally-relevant code smell" OR "hotspot pattern") AND (maintainability OR modularity OR reusability OR analyzability OR modifiability OR testability)

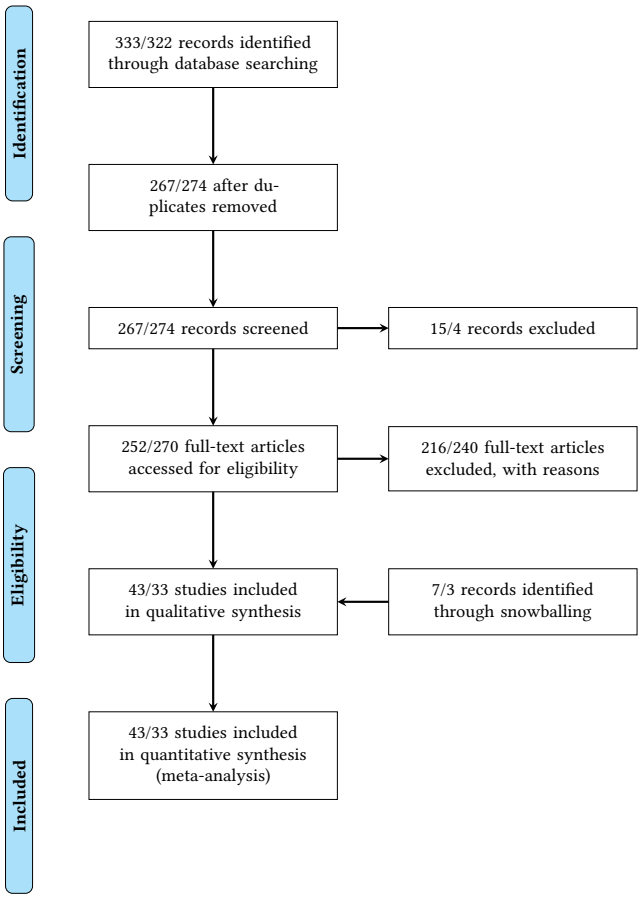

Figure 2: Adjusted flow chart for the review process [38]. The first number is for the search of connected principles, the second one for connected maintainability attributes.

The term is structured the same way as with the design principles, but we added the terms "maintainability" and its sub-characteristics to find the connected quality attributes.

We included first-tier grey literature [19] such as books and articles written by experts as they contain qualitative assertions.

We found 333 (principles) and 322 (quality attributes) articles in the earlier mentioned databases. After removing the duplicate reports, we reduced the number of unique articles to 267 and 274. The whole list is in the additional material (see Section 9.1). The inquiries were last updated on January 7th, 2022.

Study selection. We defined inclusion and exclusion criteria to select only the studies relevant to our research. The rationale of the inclusion criteria was to consider results that connected architecture smells to software design principles or maintainability and its sub-characteristics. The articles found with the two search strings were not merged but considered separately. The purpose of defining the exclusion criteria was to discard results containing insufficient information for answering our research questions.
Inclusion Criteria for the software design principles:

- principles mentioned in the context of an AS

Inclusion Criteria for the quality attributes:

- maintainability attribute named in the context of an AS

Exclusion Criteria:

- non-english

In the screening phase, we excluded one broken PDF, records not in English, and search results that were not research or written texts by experts, e.g., a call for a summer school or a table of content. The remaining 252 (principle) and 270 (QA) results were available, or the authors made them available for the eligibility phase.

We did a full-text search because the title and abstract were not sufficient to decide whether a connection to a principle or quality attribute was discussed in the article. We searched for the terms "principle" and each maintainability attribute, including maintainability itself. Then we read the paragraph and evaluated whether the text contained a concrete connection. The connection to principles or quality attributes was either just stated or argumentatively derived.

After applying these criteria, we reduced the number of articles to 36 for the software design principles and 30 for the connected quality attributes. The complete reference list of all relevant papers is in the additional material (see Section 9.1).

Verification of the search process. We verified the choice of databases as well as search strings by selecting existing studies that included connections of architecture smells and software design principles and maintainability characteristics [4, 18, 18, 28]. Our search strategy successfully identified all of these studies were successfully identified by our search strategy, indicating field coverage. In addition, we used synonymous terms for our search strings other studies already found [14, 50].

We performed a backward snowballing strategy. Here, we only screened papers that included architecture smells or a synonym in the title or were directly referenced when a relation to a design principle or quality attribute was mentioned. We found seven additional records for connected principles and three records for the quality attributes. This resulted in 43/33 included papers.

Data extraction. After an in-depth reading and analysis of the included articles, we collected the defined architecture smells and entered them into the ontology. Here, we also considered design smells that were not explicitly differentiated from architecture smells. We looked for explicit mentions of violated principles and specifications of smells that impaired maintainability or sub-characteristics. Additionally, we noted explicit relations between AS, i.e., mentions of equivalent and correlated AS.

5 RESULTS

5.1 Architecture Smells

We found 148 terms for architecture smells, including different terms for the same smell. We marked AS with the same definition as equivalent or where it was explicitly noted in the literature. We identified 114 unique architecture smells, from which 17 represent subclasses of other smells.

The ontology is too extensive to utterly present in this paper, but is included in the additional material (see Section 9.1). To illustrate

the results, we picked the AS *cyclic dependency (CD)* as a representative example. The research community does not use a unique name for this smell. We found three other terms, namely: *dependency cycle*, *circular dependency*, *cyclically dependent modularization*. In addition, we identified twelve subclasses (see Figure 3) that describe a more specific occurrence of this smell. The classes connected in both directions are equivalent, e.g., *cross package cycle* and *package cycle*.

5.2 Software Design Principles

We found 139 connections to 21 software design principles in the literature. They resulted in 58 actual entries for the ontology after removing duplicates and merging smells with the same definitions. The connections were either just stated in the literature or argumentatively justified.

Again, the overview is too extensive to be included in this paper. We will limit ourselves to a small excerpt and CD. The complete overview is in the supplementary material (see Section 9.1).

The researchers did not always use established principles such as *SOLID* [31]. Some studies used self-defined principles such as *Hierarchy* [4] and the *Principle of hierarchical structure* [7], which are based on common recommendations for good software design.

Figure 4 shows which principles *cyclic dependency* and its equivalent classes and subclasses violate. The identified connections of the equivalent classes also apply to *cyclic dependency* through the ontology. Furthermore, all connections valid for *cyclic dependency* are also valid for its subclasses. SOLID is presented as a principle even though it consists of five principles because, in one paper, they did not differentiate between the underlying principles [44].

5.3 Quality Attributes

We found 132 connections of AS to maintainability or one of the sub-characteristics in the literature. They resulted in 73 unique entries for the ontology after merging duplicates and smells with the same definitions. Table 1 shows the uncovered connections. We did not note the number of mentions because some papers were written by the same authors, which would distort the importance of the found connections. In general, the connections again were either just stated in the literature or argumentatively justified.

Cyclic dependency impairs all sub-categories, supporting that it is one of the most crucial smells [13, 33].

6 DISCUSSION

According to the goals of the knowledge base, we need information for the prioritization of architecture smells for repaying ATD. First, a precise software system analysis regarding architecture smells requires a comprehensive and clear set of AS. The AS have to have a well-established terminology and a mature classification. We discuss to which extent our results fulfill this need. Second, violations of design principles as another essential type of indicator help with the analysis regarding ATD. We discuss how they and an analysis of their relations to architecture smells contribute to the clarity, uniqueness, and efficiency of discovering ATD to answer RQ1. Third, the impact of architecture smells on maintainability characteristics is needed for prioritization. We discuss how far the studied literature provides this information to answer RQ2.

6.1 Architecture Smells

Terminology is crucial for analysis and identification, and even developer discussions about the severity and criticality of smells can hardly be performed without a clear terminology.

We found different terms for *architecture smell* such as "architectural (bad) smells" [15], "(architecture) anti-patterns" [36], "hotspot pattern" [35] and "architecturally-relevant code smells" [5]. Some are used frequently, and others, such as "architectural decay instances" [37], can only be found in one paper. Some authors even switched terms across papers [35, 36].

There may be even more terms, but they are hard to find through a search because conceivable terms such as "flaw" or "defect" are overloaded and result in many studies to analyze. Even the term "hotspot pattern" is used for an accumulation of defects and in other areas for a concentration of something, e.g., when the communication on on-chip multiprocessors leads to an overload [1]. Furthermore, different names for the same smell make identification and analysis difficult. They may be similar, such as "*cyclic dependency*" [15], "*dependency cycle*" [26], but some are utterly different, such as "first lady component" [8] and "god component" [47].

The different names for the smells sometimes derive from the different levels of abstraction. Some papers differentiate between design and architecture [45], while others do not [8]. Other researchers define smells depending on whether the smell occurred on the package-[7], subsystem-[28], module-[35] or component-level[8]. We included design smells when they were mentioned in an architectural context. We were not able to draw a clear boundary between design and architecture level because design smells can also have an architectural impact. Hence, we see the need for further investigation.

It is not clear from the study results whether this differentiation is useful for the practitioner or impacts refactoring. Especially if the abstraction level was not clearly defined and may even mean the same thing, e.g., "module" and "component". Other work shows that architecture and code smells are independent of each other [12].

Terms need to be defined to communicate the issues and support discussions about priorities. All the problems mentioned above indicate that more work still has to be done on the basic level of definitions and finding consensus about terms. Our ontology is the first step in this direction. We collected architecture terms, marked equivalent classes, and built a hierarchy where we found clear indications.

6.2 Violated Software Design Principles

To identify ATD, both architecture smells and design principles violations serve as indicators. According to our study's results, AS are still ambiguous. Therefore, violations of principles increase the precision of TD identification and contribute to more clarity regarding impact. Research shows that practitioners find these connections helpful [17].

Through the violated design principles, the causes of ATD can be identified and addressed. The design principles help assess if the occurred architecture smell is just potential or effective ATD [46] and, therefore, must be effectively dealt with. Furthermore, because they propagate quality attributes, they indicate which quality attribute

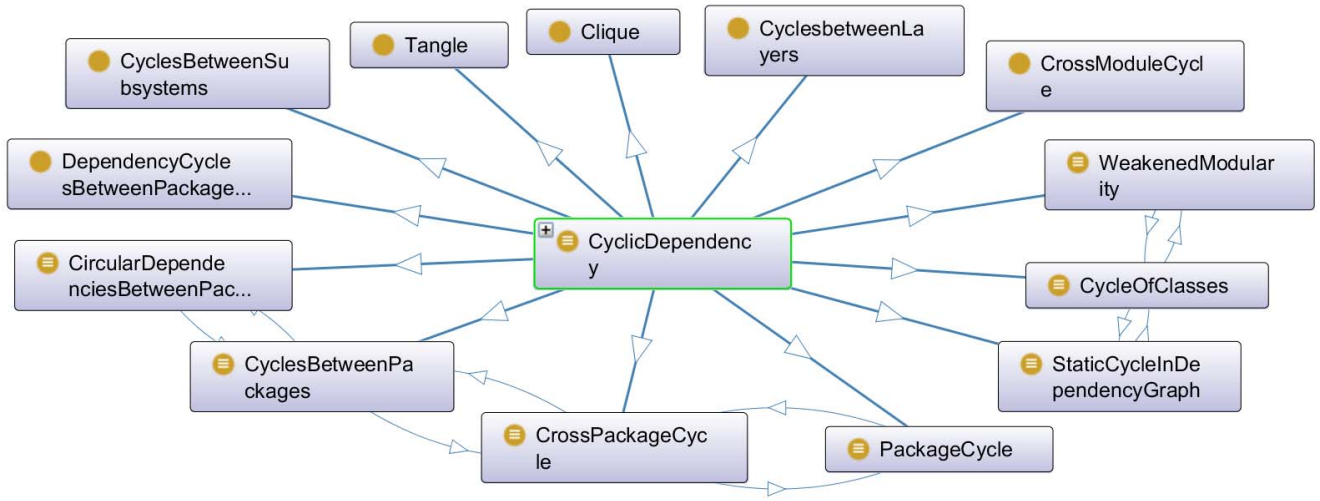

Figure 3: Subclasses of *cyclic dependency*. Classes with arrows in both directions are equivalent.

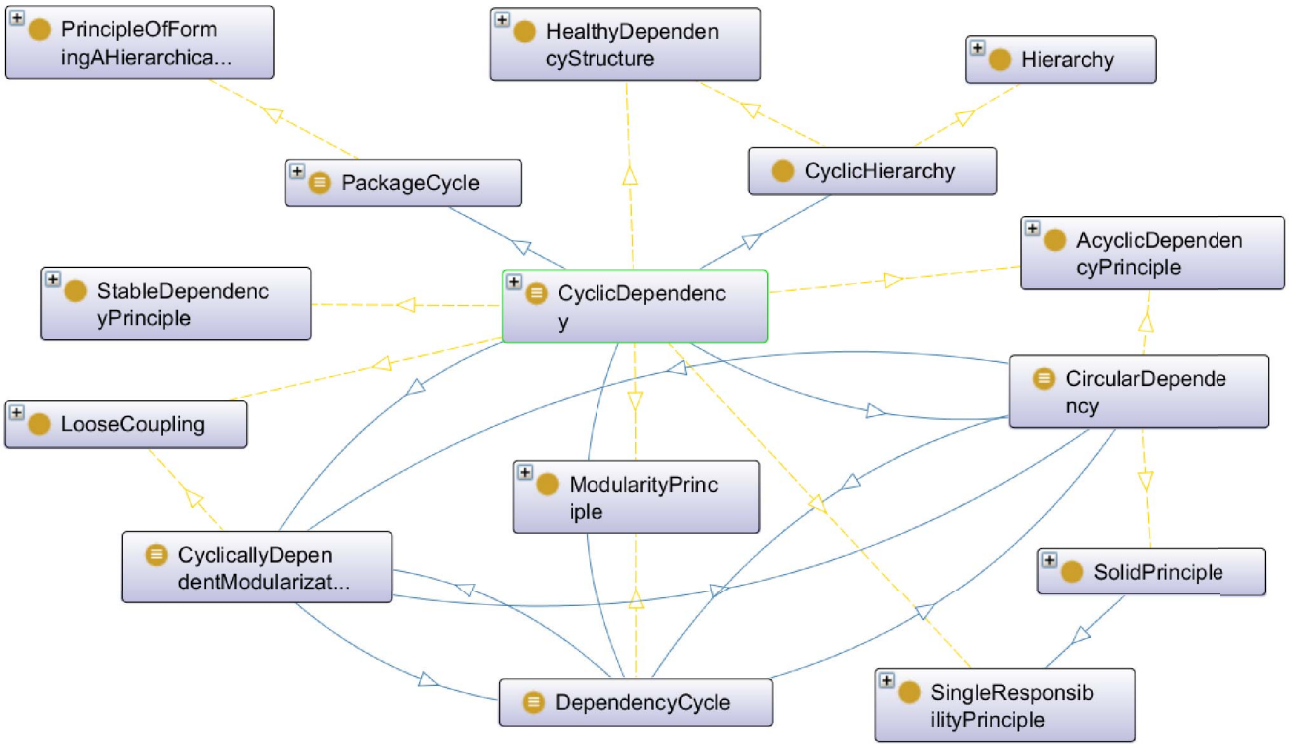

Figure 4: Violated software design principles by cyclic dependency (yellow: violates, blue: subclass of)

is impaired and make the impact of an AS more explicit. Often violated design principles can be identified with the knowledge base and addressed in the team to help prevent ATD in the future.

6.3 Impacted Quality Attributes

We have uncovered various connections (see Table 1) architects can use to prioritize the found AS according to their impact on quality

Smells	Maintainability	Analyzability	Modularity	Reusability	Modifiabilty	Testability
Ambiguous interface	x	x	x	x		
Brick functionality overload			x		x	
Broken modularization					x	
Bossy component		x		x		x
Concern overload	x	x	x			
Connector envy	x	x	x	x	x	x
Cyclic dependency	x	x	x	x	x	x
Duplicate functionality			x	x	x	
Extraneous adjacent connector		x	x	x		
Feature concentration	x	x			x	
Functional decomposition				x	x	
God component	x	x	x	x	x	x
Hub-like dependency	x	x	x	x	x	x
Insufficient package cohesion	x		x			
Lego syndrome			x	x		
Logical coupling					x	
Modularity violation groups	x		x			
Multipath hierarchy		x		x		
Over-generic design	x					
Poltergeist					x	
Scattered functionality	x	x	x	x	x	x
Sloppy delegation			x			
Unhealthy inheritance	x					
Unnecessary abstraction		x			x	
Unnecessary dependencies					x	
Unstable dependencies	x					
Unstable interface	x					
Unused interface		x	x			

Table 1: Quality attributes mentioned as impaired by architecture smells.

goals. This paper is limited to the maintainability characteristics since the internal quality attributes are essential for ATD [3].

The impaired quality attributes can be seen as consequences of an AS. Therefore, the impaired quality attributes also indicate the smell's severity. For example, the smell *cyclic dependency* is critical because it impairs all maintainability characteristics, but smells such as *unused interface* show fewer negative consequences. For the example of projects with a consistent team, smells such as *brick functionality overload* may have a higher priority than *ambiguous interface* because they may prioritize modifiability over analyzability (see Table 1).

More AS should be explicitly connected to impaired quality attributes. The results also do not allow weighting of the violations, as there were not enough unique connections in the literature, and the violations were not compared on that level. One reason for the missing impact on quality attributes might be the dependency on concrete project situations, e.g., their module structure, interface modularity, third-party components, and even the general maintainability level. Because AS are often defined as having an impact on maintainability, there are probably more AS that impact maintainability and its sub-categories.

7 THREATS TO VALIDITY

In this section, we present threats to validity based on the guidelines provided by Wohlin et al. [53].

7.1 Construct Validity

Construct validity refers to the appropriateness of the chosen methods [53]. To ensure that our approach is sufficiently defined, we carefully reviewed existing literature in the field to define a research objective and wrote a study protocol section 9.1 before conducting the literature. Furthermore, we carefully followed the tactics proposed by Kitchenham and Charters [24].

7.2 Internal Validity

Internal validity threats are related to possible wrong conclusions about causal relationships between treatment and outcome [53]. In order to exhaustively identify literature for answering our research questions and ensure that the process of papers' selection was unbiased as far as possible, we followed the guidelines by Kitchenham and Charters [24]. We used multiple databases and a broad search term. We defined the search term according to synonyms for AS we found in the literature. Although inclusion and exclusion criteria were defined and applied rigorously, only one researcher assessed

each study. This researcher's bias was mitigated to a limited extent by discussing cases that seemed ambiguous. We only included literature stating a clear connection between AS and design principles or quality attributes.

7.3 External Validity

External validity threats refer to the ability to generalize the result [53]. In our case, we used a broad search term to not limit the AS to a specific domain. A problem could be that the revealed studies did not focus on the connection of architecture smells, software design principles, and quality attributes, making the findings less reliable. To minimize this threat, we confirmed each connection by comparing the definitions of the AS and their implications. The external validity also depends on the validity of the selected studies. In our work, we were not able to evaluate the external validity of all the included studies, but we only chose peer-reviewed research or first-tier grey literature with a high outlet control and high credibility.

7.4 Conclusion Validity

Conclusion validity relates to the reliability of the conclusions drawn from the results [53]. In our case, threats relate to the potential non-inclusion of some studies. To mitigate this threat, we carefully applied the search strategy, performing the search in five digital libraries in addition to the snowballing process and cross-checking Google Scholar. We applied a broad search string, which led to many articles but included more possible results. We verified the completeness by checking the inclusion of previously manually selected studies. We defined inclusion and exclusion criteria and applied them to the full text.

8 CONCLUSION

Architecture smells are essential indicators for architecture refactorings. Prioritization of architecture refactoring is needed to repay architecture technical debt efficiently. However, AS are not yet well established and commonly used; the definitions contain ambiguity, vague relations with other smells and refactorings, and ill-defined impact on quality attributes.

To replace the current simple lists of smells, we built an extendable ontology for architecture smells that serves as a knowledge base. The knowledge base provides AS with their corresponding software design principles and maintainability attributes. By connecting this information, the ontology improves the comprehensibility of architecture smells to support decision-making on repaying architecture technical debt in current projects and builds a standardization of AS for the next future.

The knowledge base was populated by studying literature about AS. We analyzed the different names, concepts, and abstractions together with the impact on quality attributes and relations with design principles and other smells. We discussed the validity of the analysis results according to the state of the art guidelines.

9 FUTURE WORK

We are currently expanding our ontology by including detection tools and metrics to identify architecture smells. Finding connections between quality attributes, metrics, and design principles can further improve knowledge and help tackle the complex interplay of all factors. We expect that this step enhances understanding and helps to differentiate general and project-specific effects.

In the future, we plan to extend our ontology by investigating other quality attributes such as performance and security. In other research communities, other terms are more common, such as "anti-pattern" in the performance [10] and "flaw" in the security engineering community [51].

The next step involves practitioners validating the ontology and verifying the names for the AS. As there is no established catalog yet, it is questionable if the names found in the literature are self-explanatory and suitable. Furthermore, we would like to shorten the list of smells to a compact and manageable number that still comprises the key indicators, which a study could achieve. Pattern writer workshops [9] could also be used as they have proven effective and efficient for improving term maturity with practitioners' involvement.

9.1 Data Availability

The data set of the literature review has been made available as open data via Zenodo[1]. The files consists of the included studies with their classification, excluded studies with their decisive exclusion criteria, the exported OWL file of the ontology from Protégé [41, 42], the study protocol, and we also included the tables of the architecture smells and the connected design principles and quality attributes. The ontology is also made available via WebVOWL[2] [29].

REFERENCES

[1] Najla Alfaraj, Junjie Zhang, Yang Xu, and H. Jonathan Chao. 2011. HOPE: Hotspot Congestion Control for Clos Network on Chip. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip (NOCS '11)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/1999946.1999950

[2] Roberta Arcoverde, Everton Guimarães, Isela Macía, Alessandro Garcia, and Yuanfang Cai. 2013. Prioritization of Code Anomalies Based on Architecture Sensitiveness. In *2013 27th Brazilian Symposium on Software Engineering*. 69–78. https://doi.org/10.1109/SBES.2013.14

[3] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138. https://doi.org/10.4230/DagRep.6.4.110

[4] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. 2019. Architectural Smells Detected by Tools: A Catalogue Proposal. In *Proceedings of the Second International Conference on Technical Debt (TechDebt '19)*. IEEE Press, 88–97. https://doi.org/10.1109/TechDebt.2019.00027

[5] Isela Macia Bertran. 2011. Detecting Architecturally-Relevant Code Smells in Evolving Software Systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1090–1093. https://doi.org/10.1145/1985793.1986003

[6] William H Brown, Raphael C Malveau, Hays W" Skip" McCormick, and Thomas J Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

[7] Yuanfang Cai and Rick Kazman. 2019. DV8: Automated architecture analysis tool suites. In *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019*. 53–54. https://doi.org/10.1109/TechDebt.2019.00015

[8] Giuliana Carullo. 2020. *Design Smells*. Apress, Berkeley, CA, 43–57. https://doi.org/10.1007/978-1-4842-6162-0_4

[9] James O Coplien and Bobby Woolf. 1997. A pattern language for writers' workshops. *C Plus Plus Report* 9 (1997), 51–60.

[10] Vittorio Cortellessa. 2013. Performance Antipatterns: State-of-Art and Future Perspectives. In *Computer Performance Engineering - 10th European Workshop, EPEW 2013, Venice, Italy, September 16-17, 2013. Proceedings*, Maria Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin (Eds.), Vol. 8168. Springer, 1–6. https://doi.org/10.1007/978-3-642-40725-3_1

[1] https://doi.org/10.5281/zenodo.6362209
[2] https://tinyurl.com/yax69rcd

[11] Rodrigo Reboucas De Almeida, Rafael Do Nascimento Ribeiro, Christoph Treude, and Uira Kulesza. 2021. Business-Driven Technical Debt Prioritization: An Industrial Case Study. In *Proceedings - 2021 IEEE/ACM International Conference on Technical Debt, TechDebt 2021*. Institute of Electrical and Electronics Engineers Inc., 74–83. https://doi.org/10.1109/TECHDEBT52882.2021.00017 arXiv:2010.09711

[12] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. 2019. Are Architectural Smells Independent from Code Smells ? An Empirical Study. *Journal of Systems and Software* 154, April (2019), 139–156. https://doi.org/10.1016/j.jss.2019.04.066 arXiv:1904.11755

[13] Francesca Arcelli Fontana, Federico Locatelli, Ilaria Pigazzini, and Paolo Mereghetti. 2020. An Architectural Smell Evaluation in an Industrial Context. In *ICSEA 2020*. 68–74. http://thinkmind.org/index.php?view=article&articleid=icsea_2020_1_100_10042

[14] Francesca Arcelli Fontana, Ilaria Pigazzini, Claudia Raibulet, Stefano Basciano, and Riccardo Roveda. 2019. PageRank and Criticality of Architectural Smells. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2 (ECSA '19)*. Association for Computing Machinery, New York, NY, USA, 197–204. https://doi.org/10.1145/3344948.3344982

[15] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. 2017. Arcan: A Tool for Architectural Smells Detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 282–285. https://doi.org/10.1109/ICSAW.2017.16

[16] Francesca Arcelli Fontana, Riccardo Roveda, Marco Zanoni, Claudia Raibulet, and Rafael Capilla. 2016. An experience report on detecting and repairing software architecture erosion. In *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016*. IEEE, 21–30. https://doi.org/10.1109/WICSA.2016.37

[17] S. G. Ganesh, Tushar Sharma, and Girish Suryanarayana. 2013. Towards a principle-based classification of structural design smells. *Journal of Object Technology* 12, 2 (2013), 1–29. https://doi.org/10.5381/jot.2013.12.2.a1

[18] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a Catalogue of Architectural Bad Smells. *Architectures for Adaptive Software Systems* (2009), 146–162. https://doi.org/10.1007/978-3-642-02351-4_10

[19] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121.

[20] Thomas R Gruber. 1993. A translation approach to portable ontology specifications. *Knowledge acquisition* 5, 2 (1993), 199–220. https://doi.org/10.1006/knac.1993.1008

[21] ISO. 2011. ISO/IEC 25010:2011. (2011). https://www.iso.org/standard/35733.html

[22] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. 1998. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*. 68–78. https://doi.org/10.1109/ICECCS.1998.706657

[23] Ilya Khomyakov, Zufar Makhmutov, Ruzilya Mirgalimova, and Alberto Sillitti. 2020. An Analysis of Automated Technical Debt Measurement. *Lecture Notes in Business Information Processing* 378 LNBIP (2020), 250–273. https://doi.org/10.1007/978-3-030-40783-4_12

[24] Barbara Ann Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE 2007-001. Keele University and Durham University Joint Report.

[25] Duc Minh Le, Carlos Carrillo, Rafael Capilla, and Nenad Medvidovic. 2016. Relating Architectural Decay and Sustainability of Software Systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 178–181. https://doi.org/10.1109/WICSA.2016.15

[26] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2018. An empirical study of architectural decay in open-source software. In *2018 IEEE International conference on software architecture (ICSA)*. IEEE, 176–17609. https://doi.org/10.1109/ICSA.2018.00027

[27] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. 2021. A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software* 171 (2021), 110827. https://doi.org/10.1016/j.jss.2020.110827

[28] Martin Lippert and Stephen Roock. 2006. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, UK.

[29] Steffen Lohmann, Vincent Link, Eduard Marbach, and Stefan Negru. 2015. WebVOWL: Web-based Visualization of Ontologies. In *Proceedings of EKAW 2014 Satellite Events (LNAI)*, Vol. 8982. Springer, 154–158. https://doi.org/10.1007/978-3-319-17966-7_21

[30] M. Mantyla, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 381–384. https://doi.org/10.1109/ICSM.2003.1235447

[31] Robert C Martin, James Grenning, and Simon Brown. 2018. *Clean architecture: a craftsman's guide to software structure and design*. Number s 31. Prentice Hall.

[32] Antonio Martini, Jan Bosch, and Michel Chaudron. 2014. Architecture technical debt: Understanding causes and a qualitative model. *Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014* (2014), 85–92. https://doi.org/10.1109/SEAA.2014.65

[33] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. 2018. Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company. In *Software Architecture*, Carlos E. Cuesta, David Garlan, and Jennifer Pérez (Eds.). Springer International Publishing, 320–335. https://doi.org/10.1007/978-3-030-00761-4_21

[34] Steve McConnell. 2008. Managing technical debt. *Construx Inc.* (2008).

[35] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*. IEEE, 51–60. https://doi.org/10.1109/WICSA.2015.12

[36] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2021. Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1008–1028. https://doi.org/10.1109/TSE.2019.2910856

[37] Ran Mo, Joshua Garcia, Yuanfang Cai, and Nenad Medvidovic. 2013. Mapping Architectural Decay Instances to Dependency Models. In *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD '13)*. IEEE Press, 39–46. https://doi.org/10.1109/MTD.2013.6608677

[38] David Moher, Alessandro Liberati, Jennifer Tetzlaff, Douglas G Altman, and Prisma Group. 2009. Preferred reporting items for systematic reviews and meta-analyses: the PRISMA statement. *PLoS medicine* 6, 7 (2009), e1000097. https://doi.org/10.1371/journal.pmed.1000097

[39] Boris Motik, Peter Patel-Schneider, and Bijan Parsia. [n. d.]. OWL 2, Web Ontology Language – Document Overview. Retrieved January 18, 2022 from http://www.w3.org/TR/owl2-overview/

[40] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2021. A systematic mapping study on architectural smells detection. *Journal of Systems and Software* 173 (2021), 110885. https://doi.org/10.1016/j.jss.2020.110885

[41] Mark A Musen and Protégé Team. [n. d.]. Protégé. Retrieved January 18, 2022 from http://protege.stanford.edu

[42] Mark A Musen and Protégé Team. 2015-06. The Protégé Project: A Look Back and a Look Forward. *AI matters* 1, 4 (2015-06), 4–12. https://doi.org/10.1145/2757001.2757003 arXiv:27239556

[43] Natalya F. Noy and Deborah L. McGuinness. 2001. Ontology Development 101: A Guide to Creating Your First Ontology. *Stanford Knowledge Systems Laboratory* March (2001), 25. https://doi.org/10.1016/j.artmed.2004.01.014

[44] Jesper Olsson, Erik Risfelt, Terese Besker, Antonio Martini, and Richard Torkar. 2021. Measuring affective states from technical debt: A psychoempirical software engineering experiment. *Empirical Software Engineering* 26, 5 (2021), 1–40. https://doi.org/10.1007/s10664-021-09998-w arXiv:2009.10660

[45] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. 2016. Refactoring for Software Architecture Smells. In *Proceedings of the 1st International Workshop on Software Refactoring (IWoR 2016)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/2975945.2975946

[46] Klaus Schmid. 2013. A Formal Approach to Technical Debt Decision Making. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA '13)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/2465478.2465492

[47] Tushar Sharma. 2019. How Deep is the Mud: Fathoming Architecture Technical Debt Using Designite. In *Proceedings of the Second International Conference on Technical Debt (TechDebt '19)*. IEEE Press, 59–60. https://doi.org/10.1109/TechDebt.2019.00018

[48] Michael Stal. 2014. Refactoring software architectures. In *Agile Software Architecture*. Elsevier, 63–82.

[49] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[50] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. 2019. How Developers Discuss Architecture Smells ? An Exploratory Study on Stack Overflow. *Proceedings - 2019 IEEE International Conference on Software Architecture, ICSA 2019* February (2019), 91–100. https://doi.org/10.1109/ICSA.2019.00018

[51] Radu Vanciu and Marwan Abi-Antoun. 2013. Finding architectural flaws using constraints. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 334–344. https://doi.org/10.1109/ASE.2013.6693092

[52] Robert Verdeccia, Philippe Kruchten, Patricia Lago, Roberto Verdecchia, Philippe Kruchten, and Patricia Lago. 2020. Architectural Technical Debt : A Grounded Theory. In *European Conference on Software Architecture (ECSA 2020)*, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann (Eds.). Springer, Cham., 202–219. https://doi.org/10.1007/978-3-030-58923-3_14

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media. https://doi.org/10.1007/978-3-642-29044-2

[54] Olaf Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (2015), 26–29. https://doi.org/10.1109/MS.2015.37