How Developers Discuss Architecture Smells? An Exploratory Study on Stack Overflow

Fangchao Tian [a,b], Peng Liang [a*], Muhammad Ali Babar [b]
[a] School of Computer Science, Wuhan University, Wuhan, China
[b] School of Computer Science, The University of Adelaide, Adelaide, Australia
tianfangchao@whu.edu.cn, liangp@whu.edu.cn, ali.babar@adelaide.edu.au

***Abstract*—Architecture Smells (ASs) are design decisions that can have significant negative effects on a system's quality attributes such as reusability and testability. ASs are focused on higher level of software systems than code smells, which are implementation-level constructs. ASs can have much wider impact on a system than code smells. However, ASs usually receive less attention than code smells in both research and practice. We have conducted an exploratory study of developers' conception of ASs by analyzing related discussions in Stack Overflow. We used 14 ASs related terms to search the relevant posts in Stack Overflow and extracted 207 posts. We used Grounded Theory method for analyzing the extracted posts about developers' description of ASs, causes of ASs, approaches and tools for detecting and refactoring ASs, quality attributes affected by ASs, and difficulties in detecting and refactoring ASs. Our findings show that: (1) developers often describe ASs with some general terms; (2) ASs are mainly caused by violating architecture patterns, design principles, or misusing architecture antipatterns; (3) there is a lack of dedicated tools for detecting and refactoring ASs; (4) developers mainly concern about the maintainability and performance of systems affected by ASs; and (5) the inability to quantify the cost and benefit as well as the lack of approaches and tools makes detecting and refactoring ASs difficult.**

Keywords—Architecture Smell, Stack Overflow, Architecture Refactoring, Quality Attribute

I. INTRODUCTION

When software systems evolve, some bad smells are identified and fixed to maintain the systems. These smells can be classified into three categories according to their granularity: Architecture Smells (ASs), design smells, and code smells [2]. All these types of smells can negatively affect a system's quality with different scopes of impact. Code smells are design issues that degrade understandability and changeability at the code level, which results in poor maintainability and hinders evolution of a system [4]. Design smells indicate violation of design principles such as acyclic dependencies principle, and have a potential impact on a set of classes in design structure [5]. ASs are the higher level design issues of a system. ASs are a typical type of architectural technical debt and can negatively affect system-wide maintainability of a software system. Refactoring ASs usually costs more effort than refactoring code smells and design smells [33].

During the evolution of a software system, it is important to periodically identify and refactor various types of smells to improve evolution and maintenance of software quality. Hence, refactoring is regarded as an integral part of software development process. Each smell has corresponding refactoring to improve software quality at different granularity levels. Code refactoring is carried out to eliminate code smells. Architectural refactoring, akin to code refactoring, is used to remove ASs [2].

Many studies concentrate on investigating code smells. Fowler [3] presented 22 kinds of code smells in his book. Beyond that, Brown *et al.* and Moha *et al.* identified other code smells, such as Spaghetti Code [6] and Swiss Army Knife [7]. Also, many techniques and tools are employed to detect and refactoring code smells, such as JSpIRIT [11], PMD [12], and Jdeodorant [13]. Gupta *et al.* [14] conducted a review to investigate code smells in java source code, and he concluded that only five code bad smells are not provided with associated detection techniques in the literature.

Unlike code smells, there is limited literature published on ASs. There is no general consensus on the definition of ASs as different approaches define ASs differently. Samarthyam *et al.* [2] identified various definitions and terms for ASs, such as architectural bad smell [8], architectural defect [15], architecture anti-pattern [6], and architecturally-relevant code smell [16]. Despite that researchers proposed a catalogue of ASs and corresponding architectural refactorings [8], it is not clear how well their definitions of ASs are accepted and to what extent their refactoring approaches and tools are adopted by practitioners when detecting and refactoring ASs. There is a lack of empirical studies focusing on practitioners' perception of ASs and how they detect and refactor ASs. To this end, this paper reports a study of how developers discuss ASs in practice.

As one of the online software development communities, Stack Overflow has been a most popular and widely used questions and answers (Q&A) platform for developers to ask and answer questions since 2008 [9]. Stack Overflow covers a wide range of topics of software development as a crowdsourced knowledge sharing platform. Recent studies have used Stack Overflow to extract architecture-relevant knowledge such as architectural decisions [21] and architecture patterns [24]. Tahir *et al.* investigated code smells and antipatterns over Stack Overflow [29]. We have used Stack Overflow as the source to search and extract 207 posts. We used Grounded Theory [32] to analyze the extracted posts for the purpose of: (1) identifying the terms developers use to describe ASs, (2) classifying the causes

* Corresponding author
This work is partially funded by the NSFC under Grant No. 61472286 and the China Scholarship Council.

978-1-7281-0528-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA.2019.00018

IEEE computer society

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

of ASs and the approaches and tools to detect and refactor ASs as well as the difficulties in detecting and refactoring ASs, and (3) collecting the quality attributes affected by ASs. This study is aimed at gaining evidence-based knowledge about developers' perception of ASs.

The rest of the paper is structured as follows. Section II describes the related work. The research methodology (including research questions) is detailed in Section III. Section IV provides the study results, and Section V discusses the implications of the results. The threats to validity are presented in Section VI. Section VII concludes this work with future directions.

II. Related Work

In this section, we review the literature on ASs and the use of Stack Overflow for empirical studies. We first present the studies about the definitions of ASs as well as the approaches and tools for detecting and refactoring ASs. Next, we briefly review the works that focus on addressing software engineering problems using developers' discussions on Stack Overflow.

A. Architecture Smells

Several studies proposed the definition of AS with specific subtypes. Lippert *et al.* [17] treat ASs as a type of bad smells indicating an underlying problem at a higher level of a system's granularity. Garcia *et al.* [8] propose the commonly used definition of AS "*a commonly (although not always intentionally) used architectural decision that negatively impacts system quality*". This definition was adopted by many studies, such as Amirat *et al.* [18], and *Kaur et al.* [19], which investigated AS refactoring using a graph transformation approach and assessed ASs in product line architecture respectively. These papers pointed out that it is critical to detect and refactor ASs for maintaining system design.

Le *et al.* [10] described four different categories of ASs: interface-based smells, change-based smells, concern-based smells, and dependency-based smells. They also proposed corresponding architecture decay metrics to detect these ASs and suggested quality attributes affected by ASs. Samarthyam *et al.* [2] discussed the necessity of architectural refactoring, and many approaches and tools had been proposed to detect and refactor ASs. A tool named Arcan was developed by Fontana *et al.* to detect three kinds of ASs based on evaluating architecture dependency graph [20][33]. Rizzi *et al.* presented a tool (as an extension of Arcan) that identifies and removes Cyclic Dependency smells [34]. Mo *et al.* [35] formally defined five architecture hotspot patterns (a type of ASs) and reported a hotspot detector to automatically detect these five ASs in code base. Bertran *et al.* [16] defined a set of architecturally-relevant code smells, and analyze the techniques for detecting design smells.

Our literature review indicates that there is no reported empirical research on ASs especially from the perspective of developers. To close this gap, we have conducted an exploratory study to investigate developers' understanding of ASs and the refactoring tools used.

B. Empirical Studies using Stack Overflow

Recently, researchers and practitioners have been making use of developers' questions, answers, and discussions available on Stack Overflow for exploring a wide range of topics since the first paper was published in 2011 [22]. Zou *et al.* [25] employed the topic model technique to analyze non-functional requirements-related textual content on Stack Overflow posts with the goal of comprehending the actual requirements of developers. Venkatesh *et al.* [26] reported a Python crawler to extract discussions about Web APIs and applied Latent Dirichlet Allocation to analyze the topics of Web API. Similarly, Linares-Vásquez *et al.* [27] and Yang *et al.* [28] used topic modeling techniques to obtain the topics of mobile-development related questions and security-related questions respectively. Tahir *et al.* [29] investigated developers' perceptions of code smells and antipatterns by analyzing the related posts on Stack Overflow. Bi *et al.* [24] analyzed architecture patterns and their relationships with quality attributes and design contexts by collecting architecture pattern related discussions on Stack Overflow. In this study, we aim to extract the discussions of ASs from Stack Overflow with the aim of studying developers' perception of ASs.

III. Methodology

We aimed at revealing the insights of developers about the characteristics of ASs and their impact on software systems as well as development effort. The goal of this study is: to explore the definitions and classifications of ASs and corresponding refactorings; to analyze how developers detect and refactor ASs using specific approaches and tools; and to understand the causes of the occurrence of ASs and their impact on system qualities. We have conducted an exploratory study following the guideline [1] by mining and analyzing Stack Overflow posts to achieve the above goal.

A. Research Questions

According to the goal of this study, we formulate five Research Questions (RQs) in TABLE I.

TABLE I. Research Questions and Their Rationale

Research Question	Rationale
RQ1: How developers describe ASs in software development? **RQ1.1**: What is the classification of ASs according to their descriptions?	Researchers and practitioners often use different terms to describe ASs, e.g., architectural antipatterns, design smells, and architectural anomalies [2]. We want to provide not only a category of terms for describing ASs, but also want to gain an insight into how many kinds of ASs are described on Stack Overflow, in order to facilitate the understanding of the concept of AS.
RQ2: What are the causes of ASs from the developers' perspective?	We want to know why ASs are generated. The answer of this RQ can inspire developers to come up with (1) solutions for preventing ASs from happening and (2) approaches to identifying ASs.
RQ3: What approaches or tools are used to detect and refactor ASs?	We want to know how developers detect and refactor ASs.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

RQ4: What system quality attributes (such as testability and reusability) are affected by various ASs?	We want to make clear the impact of ASs on software development, in order to take appropriate actions to handle ASs.
RQ5: What are the difficulties in detecting and refactoring ASs?	When developers plan to detect and refactor ASs, they make trade-offs among many factors. We want to classify the factors that challenge developers in detecting and refactoring ASs.

B. Data Collection and Filtering

Stack Overflow[2], as a Q&A platform, is popularly used by millions of professionals (e.g., developers or architects) to post questions and collect feedback [30]. Crowdsourced knowledge on Stack Overflow provides rich information for mining and analyzing discussions on architecture smells.

(1) *Search and collect candidate posts:* We used automatic search to find relevant posts on Stack Overflow. To retrieve as many relevant posts as possible, we collected terms about ASs from literature (e.g., [2][8]). Smells in software development include defects and violations [2], thus "*architecture violation*" and "*architecture defect*" should be considered as search terms. In addition, smells are also called bad smells [8], and both "*architecture smell*" and "*architecture bad smell*" should be considered as search terms. The search terms used in this study are listed below:

"*architecture smell*" *OR* "*architectural smell*" *OR* "*architecture bad smell*" *OR* "*architectural bad smell*" *OR* "*architecture defect*" *OR* "*architectural defect*" *OR* "*architectural violation*" *OR* "*architecture violation*" *OR* "*architectural refactoring*" *OR* "*architecture refactoring*" *OR* "*architecture antipattern*" *OR* "*architectural antipattern*" OR "*architecture anti-pattern*" OR "*architectural anti-pattern*".

We used the above terms to search in the titles, questions, comments, answers, and tags of posts in Stack Overflow. Most of the existing work did not consider comments when mining and analyzing data in Stack Overflow posts (e.g., [24][27]), which may result in missing of relevant information in comments. In this study, we included comments for data collection in order to make sure that the overall contexts of the posts are considered. The automatic searches were performed by searching each term and finally we retrieved 5950 posts as listed in TABLE II. The retrieved Stack Overflow posts were sorted with the best match to the search terms.

TABLE II. RETRIEVED AND SELECTED POSTS OF SEARCH TERMS

#	Search term	No. of retrieved posts	No. of selected posts
ST1	architecture smell	339	35
ST2	architectural smell	340	12
ST3	architectural bad smell	89	26
ST4	architecture bad smell	88	3
ST5	architecture defect	103	13
ST6	architectural defect	100	1
ST7	architectural violation	997	30
ST8	architecture violation	1041	10
ST9	architecture antipattern	67	28
ST10	architectural antipattern	72	0
ST11	architecture anti-pattern	404	28
ST12	architectural anti-pattern	308	6
ST13	architectural refactoring	924	10
ST14	architecture refactoring	1078	5
Total No. of posts		5950	207

[2] https://stackoverflow.com/

(2) *Filter candidate posts:* The inclusion and exclusion criteria of post filtering are specified below: If one post contained any data which can be extracted according to the data items defined in TABLE III, a post was included for further analysis. If one post was associated with other types of smells instead of ASs, such as code level smells, this post was excluded.

Before the formal post filtering, a pilot post filtering was conducted by the first author with 50 posts randomly selected from 5950 posts, and the second author checked the results so that the two authors can get a consensus on the understanding of the abovementioned inclusion and exclusion criteria. Any disagreements and misunderstandings on the filtered results were discussed between the two authors.

The retrieved posts in TABLE II were manually checked by the first author to confirm whether the posts are related to ASs. The first author manually collected the first 50 retrieved posts by each search term, which ended up with 700 posts in total, and he removed the duplicated posts resulting in 400 posts in total. The first author then selected the posts to be further analyzed based on the abovementioned criteria. When the first author had any posts that he could not decide, these posts were discussed by the first two authors to reach a consensus. After that, we finally got 207 posts for data extraction and analysis, and the final numbers of the selected posts by each search term are shown in the last column of TABLE II.

C. Data Extraction and Analysis

(1) *Extract data:* To answer the research questions formulated in the above section, we extracted the related data from the selected 207 posts. The extracted data items to be extracted are listed in TABLE III.

TABLE III. DATA ITEMS TO BE EXTRACTED AND RELEVANT RESEARCH QUESTIONS

#	Data Item	Description	RQ
D1	Description of AS	Descriptions of ASs by practitioners based on their understanding	RQ1
D2	Cause of ASs	Causes that lead to ASs	RQ2
D3	Approach for detecting and refactoring ASs	Methods used to detect/refactor specific ASs (e.g., machine learning based approaches)	RQ3
D4	Tool for detecting and refactoring ASs	Tools for detecting and refactoring specific ASs (e.g., source code analysis tools to identify dependency cycles)	RQ3
D5	Impact of ASs	Impact of ASs on software development (e.g., understandability, testability, extensibility, and reusability)	RQ4
D6	Challenge of detecting and refactoring ASs	Challenges identified in detecting and refactoring ASs	RQ5

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

(2) *Analyze data:* We employed an interpretive research method - Grounded Theory [32] to code and analyze the extracted data for addressing RQs. Grounded Theory is a systematic bottom-up approach to generate concepts, categories, and theory from the collected qualitative data. We employed the first two coding phases (open coding and selective coding) in Grounded Theory. A qualitative data analysis tool MAXQDA[3] was used to support the coding.

Before conducting the formal coding, we ran a pilot data extraction and analysis by randomly selecting 50 posts from the 207 posts. The first author extracted the data from the posts and coded the data using Grounded Theory, and the second author checked the results. Any divergences of the extraction and coding results were discussed between the two authors in order to reach a common understanding on the extracted and coded data. In the formal data extraction and coding, the first author extracted data according to the data items (see TABLE III) from the posts and codified the data using Grounded Theory. When the first author was not sure about the extraction and coding of certain posts, the posts were discussed between the first two authors in order to eliminate the personal bias. All the data of this study has been publicly available online[4], including the selected posts in MS Word and the encoded data in MAXQDA.

IV. RESULTS

A. Results of RQ1

To answer this RQ, we used Grounded Theory to codify the data of data item D1 extracted from the 207 posts. We identified and analyzed how developers discussed and described ASs when they were aware of certain types of ASs found in their design. Most of the data about RQ1 were extracted from the titles and questions of the posts since developers tended to give descriptions of ASs by asking questions like "*Does my architecture smells bad?*". Some developers did not know what is wrong with their architectures. They asked for help in questions and respondents provided the descriptions of ASs in comments and answers of the posts, in which case we extracted the data from the comments and answers.

We identified the terms used to describe ASs by developers as listed in TABLE IV. We classified the terms used for AS descriptions into three types: Architecture, Design, and Architecturally-relevant code smell. For the architecture related terms, developers used both specific and general terms to describe ASs. Some developers used general terms to reveal that the architectures in use have certain types of smells without referring to specific ASs. They usually described ASs by using terms such as "*bad smell in architecture*", "*messy architecture*", and "*architectural defect*". For example, one developer mentioned that "*A render engine shouldn't be tied to a particular view. Yours seem like a mix of a view, view template and view renderer, it is just a bad architecture*". Some other developers used specific terms to describe ASs, such as "*violation of architecture patterns*" and "*component problems*". For example, with respect to component problems in architectures, one developer listed several specific types of ASs, such as *circular dependency among components*, *scattered parasitic functionality*, and *ambiguous interfaces*. Furthermore, developers also used terms about implementation architectures to express ASs. For example, the code architecture has defects or the architecture has bad code smells. Finally, developers do not always distinguish between design smells and ASs. They used terms like "*design defect*" or "*design smell*" to refer to AS, for example, one developer mentioned that "*DTOs by nature are easily serialized and ready to be sent to an external system in an SOA architecture or embedded into a message. Using ViewModels will have a negative impact on my architecture. In a CQRS, I suppose that is a design smell*".

B. Results of RQ2

For this question, we manually analyzed the contents of the posts to understand the context in which ASs occurred. After understanding the potential reasons for generating ASs, the related data for answering this RQ were codified.

In TABLE V, we listed 3 types of causes of ASs with their examples and occurring counts.

TABLE IV. TERMS THAT DEVELOPERS USED TO DESCRIBE ARCHITECTURE SMELLS

Type	Subtype	Term	Example	Count
Architecture	General terms	Bad smell in architecture	*The bad smell in a component architecture is if you are spending time on thinking or working on the infrastructure to manage and combine components, rather than the components themselves.*	23
		Wrong with architecture	*But how I resolve an IMaquinasListaMaquinaViewModel letting it know which Maquina is it about? What is wrong with my architecture? Because I feel like something smells bad.*	6
		Architecture defect	*I was trying the same case using Mode.objects.create() but failed as it also updates existing records. It seems to be defect in django ORM architecture.*	5
		Messy architecture	*But somehow, this smells like it should be refactored somehow. Internal is always something I'd use sparingly because it can lead to a very messy architecture.*	2
		Poor architecture	*Technical debt (also known as design debt or code debt) is a neologistic metaphor referring to the eventual consequences of poor software architecture and software development within a codebase.*	2
		Suboptimal architecture	*Thus, the use of short executable-producing iterations focused on exposing a suboptimal and/or brittle architecture.*	2
		Mistake in architecture	*Anti-pattern is not a code smell. It is a mistake in architecture.*	2
		Architectural problem	*I am currently working on a project with some (in my opinion) architectural problems.*	2
	Specific terms	Violation of architecture	*Violation of MVP architecture pattern, violation of the RESTful architecture, violation of MVC architecture, webservice architecture smell, microservice architecture smell*	32

[3] https://www.maxqda.com/

[4] https://tinyurl.com/yc3wwton

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

		patterns		
		Architectural antipattern	*Service Locator antipattern, SQL entity-attribute-value antipattern, anemic domain model, Flux antipattern*	29
		Layer problem	*Violated layer, missing layer.*	11
		Component problem	*Cyclical dependency among components, scattered parasitic functionality, ambiguous interfaces*	8
Architecturally-relevant code smell		Architecture smells in code	*I'd recommend you to inject FirstFile into SecondFile. Now your code has a smell of bad architecture.*	3
		Code smells in architecture	*N. B. This architecture has a really bad code smell. I'm sure there's a better way to do this.*	2
		Code architecture defect	*I want to know if I am working wrong, or my code architecture has defects. Here are my functions to use.*	2
Design		Bad design	*I want to mention that you should not write all your networking and business logic in your service layer. This also can be considered as a bad design.*	10
		Design defect	*There was a bug in the architecture which we encountered only through prototyping and testing. This is a design level defect completely expressible in logic, precisely the type of defect Alloy is designed for.*	7
		Design smell	*The GUI is event based, so unless it somehow just wakes up every few milliseconds and seeks out the last IDebugTarget and registers one of its views as a listener, I don't see how this is going to work. And that would be a bad design smell in Eclipses plugin architecture.*	4

(1) *Architecture antipattern.* Among the mentioned architecture antipatterns, anemic domain model has the highest frequency followed by Service Locator antipattern and Singleton antipattern. For example, developers discussed Anemic domain model in the answer of a post "*If you treat your DTOs as a domain model layer (in other words, you have no domain logic separate from the user interface), then you are using your DTOs as an anemic domain model, which is a severe architectural anti-pattern*". In most cases, when discussing these antipatterns, developers had an inconsistent understanding of whether Service Locator and Singleton are antipatterns in certain context. One developer reported "*View controllers are often created when needed and automatically deallocated when not needed anymore; that kind of view controller couldn't be a singleton. If your view controller was a singleton I would question your decision. On the other hand, for a Model class, being a singleton wouldn't be unusual at all*".

(2) *Violation of architecture pattern.* When employing architecture patterns in systems, ASs were caused due to misusing or violating these architecture patterns. For example, when using MVC pattern, developers often suffered smells due to certain violations, such as mixing of view and controller logic, leaking domain objects into view, coupling the model and view, and views created inside a controller of the class.

(3) *Violation of design principle*. If developers induced architecture smells due to violation of design principles, we consider these factors as the causes of ASs. As presented in TABLE V, the most discussed causes of ASs are violating the principle of low coupling and high cohesion, and SOLID. For violation of the low coupling principle, Circular Dependency is the most mentioned cause of ASs that violates the Acyclic Dependencies Principle. For example, one developer stated "*I have two services A and B. I use service A inside service B and service B inside service A*". Besides, violating the Separation of

TABLE V. CAUSES OF ARCHITECTURE SMELLS

Type	**Cause**	**Example**	**Count**
Architecture antipattern	Anemic domain model	*That separates the domain logic from the domain objects, leading to an anemic domain model. But that goes against all my OOD instincts; I agree with Martin Fowler that it is an anti-pattern.*	17
	Service Locator antipattern	*The Service Locator is considered an anti-pattern in modern application architecture.*	12
	Singleton antipattern	*The problem with angular's service "pattern" is the fact that it is just a singleton (due to way DI works in angular). And building your application architecture using singleton instance objects will result in naive application architecture.*	11
	Microservice antipattern	*You have one big file/db with all the parameters, settings and preferences. Your customers are used to go to this one "place" and set the system. With microservices architecture, this "centralism" seems like an antipattern.*	2
	Other antipatterns	*You can read my HYPERLINK https://vladmihalcea.com/the-open-session-in-view-anti-pattern/ Open Session In View Anti-Pattern article. Otherwise, here's a summary for why you shouldn't use Open Session In View. Open Session In View takes a bad approach to fetching data.*	10
Violation of architecture pattern	Violation of the MVC pattern	*Mixing of view/controller logic here; leaking domain objects into view, not one-view-one-view model; it's not a good way to access the current_user in a model, this logic belongs to the controller; coupling the model and the view.*	15
	Violation of Layered architecture	*found lots of architectural layering issues where lower layers are referencing upper layers.*	10
	Violation of RESTful architecture	*Duplicate validation logic on client (javascript) and server side; REST is great for*	8

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

		resources. It's really painful to map a checkout procedure as a restful resource and not worth the effort.	
	Violation of Microservice architecture	*In this regard, I consider having one common database for all services and it does violate the very essence of Microservice architecture.*	5
	Violation of Clean architecture	*That violates the dependency rule as this introduces a compile time dependency from "gateways" circle to "frameworks" circle in Clean architecture.*	2
	Others	*Violation of Sharp architecture; violation of twelve-factor architecture.*	2
Violation of design principle	Violation of low coupling and high cohesion	*Very tight integration between the swing UI and the business logic. Refactoring is a delicate and time-consuming project.*	21
	Violation of the SOLID principles	*The names AccountService and ProductService imply that you are violating the Single Responsibility Principle, Open Closed Principle and Interface Segregation Principle, which is 60% of the SOLID principles in an ASP.NET MVC app.*	15
	Violation of the SoC principle	*All the examples of MVC projects I've found have very poor separation of concerns, and we want to make sure we do this thing right.*	5
	Violation of Don't Repeat Yourself (DRY)	*I feel like I really violate the DRY principle, and I'm not feeling great with the interfaces in architecture I made that looks so similar.*	4
	Violation of database design principles	*Also, please, note, that in common case building one query by another query is an architecture smell (because of unpredictable SQL length, at least).*	3
	Violation of the KISS principle	*I'd say KISS certainly applies to architecture and design. Over-architecture is a common problem in my experience, and there's a code smell that relates: Contrived complexity; forced usage of overly complicated design patterns where simpler design would suffice.*	2
	Other principles	*Not applying clean code principle in code architecture; violation of Law of Demeter in Interactors/Use Cases/Application Services.*	12

Concern (SoC) principle also regularly triggers ASs. For example, developers mentioned that "*Architecture component A has many functionalities per se, or component A and B have many overlapping functionalities in use*".

In addition, developers hardly distinguished between architecture antipatterns and design antipatterns, as well as architecture patterns and design patterns, for example, some posts that discussed architecture patterns, but were labeled with the tag of design pattern. In our coding and analysis, only when developers mentioned design antipatterns, design patterns, and design principles in an architecture context, we considered design antipatterns, and violations of design patterns and principles as the causes of ASs.

C. Results of RQ3

Our analysis of the data reveals that developers proposed most approaches for detecting and refactoring ASs based on the abovementioned causes of ASs. Therefore, most approaches presented in TABLE VI correspond to the abovementioned causes in TABLE V. There are still cases in which certain types of ASs were mentioned without providing specific solutions, such as violating the KISS and DRY principles.

Another finding is that developers mentioned several general approaches for detecting and refactoring ASs. For example, some developers proposed to combine tools and tests in their solutions "*But a combination of tools and tests often give enough hints on various ASs*''. Beyond that, other developers came up with architecture visualization by using the Dependency Structure Matrix (DSM) to detect ASs, such as circular dependency. Developers also suggested that formal methods can be employed to specify and check the constraints and invariance of architecture for detecting ASs.

We enlist the tools developers used to detect and refactor ASs in Fig. 1 ranked by their numbers of mentions. There are no

TABLE VI. APPROACHES FOR DETECTING AND REFACTORING ARCHITECTURE SMELLS

Type	**Approach**	**Example**
Refactor architecture antipattern	Remove Entity-Attribute-Value antipattern	*You're using a variation of the terrible antipattern called Entity-Attribute-Value. Eventually you'll want a few more attributes and you might exceed some architectural limit of the database on the number of joins it can do. The solution is: don't reconstruct the row in SQL. Instead, query the attributes as multiple rows, instead of trying to combine them onto a single row.*
	Remove Service Locator antipattern	*If anyone is interested, I found a way to remove the ServiceLocator in my objects and still dynamically load/discover dependencies at runtime. The way I solved it was by registering my components in my DI container in the following way (using the Mediator pattern).*
	Remove Singleton antipattern	*This can make sense if you want to use different customisations of the APIClient for particular service classes, but if you, for some reasons, don't want extra copies or you are sure that you always will use one particular instance (without customisations) of the APIClient - make it a singleton, but DON'T, please DON'T make service classes as singletons.*
	Remove A-Session-Per-Operation antipattern	*(Topic: Refactor a DAL (Data Access Layer) with Hibernate, DAO (Data Acess Object), service and unit of work: how to put all together) DAOs should be about persistence only. They need not know anything about sessions. Let the service give the DAO what it needs to persist an object.*
	Remove Anemic domain model	*For more info look at the Anemic and Richdomain models. Some service methods and business logic can be handled in the model, so it will be a "rich" (with behaviour) model.*
Follow architecture pattern	Follow the MVC pattern	*One-view-one-viewmodel; Code logic (model) and your view (template) logic are separate; The presenter should bind the model and view in MVP; Use a subclass of UIView in viewDidlaod; Don't combine the controller, a model or a view in one class; View entirely independent of the model.*
	Follow Layered architecture	*Separation required between the layers; Construct a business layer between GUI and DAL; Business Logic runs independently of Database; Put the LINQ query into a component that feeds the UI layer.*
	Follow RESTful architecture	*REST services shouldn't use WSDLs; Stop trying to shoehorn every aspect of REST into every*

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

		application; not to duplicate validation logic on client (javascript) and server side.
	Follow Clean architecture	*The dependency rule says only dependencies from outer circles to inner circles are allowed.*
Follow design principle	Low coupling and high Cohesion	*Resolving circular dependency: decompose a system to identify cyclical dependencies; Services don't have coupling between them; keep your view and data decoupled.*
	Follow the SOLID principles	*Following the SPR: Having a number of services which adhere to the Single Responsibility Principle makes sense.*
	Follow the SoC principle	*Use your EF Model in the presentation layer but still satisfy the separation of concerns principle.*
	Follow database design principles	*Data access code isn't put in a base class; hash values should be consistent for the life of my immutable objects to solve this problem of database architecture.*
	Resolve Dependency Injection (DI) issues	*Manage dependencies with a good DI framework; Each IoC (Inversion of Control) has a way of resolving a dependency, all you need is a reference to the IoC container.*
Other approaches	Combination of tools and tests	*But a combination of tools and tests often give enough hints on various smells.*
	Experience about architecture	*Peer review or inspection of architecture smells; Experience to understand each smell's significance and impact.*
	Reassessment of the architecture	*You should seriously think back about the architecture of your solution.*
	Refactor code smells for architecture	*You can find further discussion at this question on programmers stackexchange regarding code smells for architecture.*
	Maintain conceptual integrity	*One thing that can help is for people from the original development team to be involved in architecture maintenance.*
	Architecture simplification	*Strip the architecture down to some kernel (e.g., remove use sets instead of more complicated data structures, use sig instead of more detailed enumerations)*
	Specify constraints and invariance of architecture	*I've used lightweight formal methods informally to check the constraints and invariance of the architecture.*
	Architecture visualization	*This will ensure that you have an updated dependency structure matrix (DSM) to continuously monitor the architecture of your software; Both tools support static visualizing dependencies between assemblies and classes through a DSM.*

dedicated tools that were specifically built for detecting or refactoring ASs. To summarize the results, most developers tend to analyze and assess architecture as well as identify architecture violations by investigating the code and using code analysis tools, such as PMD, NDepend, and SonarQube. On the one hand, they used Sonar architectural rule engine or PMD rule extension installed on the Sonar server to detect architecture violations. They employed NDepend that shows dependency diagrams of assemblies which can immediately track the architecture violations and take actions reactively.

Developers also created and configured their own detecting and refactoring rules for their architecture as well as gathered information about the quality of architecture with the help of the above tools. For example, some developers used Checkstyle or PMD with Eclipse IDE to flag architecture violations at the build time by configuring their own rules, as one developer stated "*We use both Lattix and NDepend to track dependencies of our assemblies. Both tools support static visualizing dependencies between assemblies and classes through defining a dependency structure matrix (DSM). A DSM gives you the ability to show the architecture of your application. For example, if you use layering this should be visible in the DSM. Cyclic dependencies will also be visible in a DSM".*

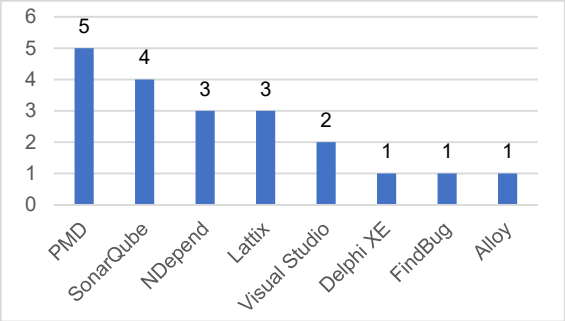

Fig. 1. Tools for detecting and refactoring architecture smells

D. Results of RQ4

TABLE VII shows the quality attributes that developers discussed about the effect of ASs with their examples and numbers of mentions. The majority of developers (69.2%, 27 out of 39 mentions) complained that smells in architecture create future maintenance pain and make software systems hard to maintain. Several developers considered that ASs may have negative effects on a software system quality, such as not achieving the performance they were looking for. Few developers mentioned that smells avoided in original architecture may creep up in the new architecture, and this situation made code hard to test and decreased code reusability. The result shows that when developers discuss ASs, they tend to take maintainability as the key consideration.

TABLE VII. QUALITY ATTRIBUTES AFFECTED BY ARCHITECTURE SMELLS WITH THEIR EXAMPLES AND COUNTS

Quality attribute	Example	Count
Maintainability	*You basically picked every antipattern I can think of - maintenance nightmare, overdesigned with tons of useless technologies in there. You force layer technologies into a tiered architecture.*	27
Performance	*There is something we could consider anti-patterns when using JMS/Asynchronous Messaging architectures? Sometimes, designs may lead us to prove side effects instead of achieving that performance we are looking for.*	5
Testability	*However, we had instances where people directly imported the model in view and called the functions violating the norms and hence the test cases couldn't be written properly.*	4
Reusability	*The main issue I see is that child components are hardly reusable in such scenario since they are tightly coupled with*	3

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

	parent component. So that's why injecting service with data is more preferable and easier to test with unit tests.	

E. Results of RQ5

We collected and classified difficulties in detecting and refactoring ASs into five types as shown in TABLE VIII. Most developers are concerned about the time cost involved in detecting and refactoring architecture. Before they made decisions on whether or not to refactor an AS, they usually made a cost-benefit analysis, but it is difficult to make sure that refactoring would speed the development and deliver a long-term benefit. Many developers stated that they lacked tools to fix ASs and evaluate the potential cost. Developers do not have the approaches to evaluate how good their architecture is and restrict subsequent developers from violating architecture patterns. Moreover, there are no definitions of what is a robust architecture and specific ASs, which hinders developers from detecting and identifying specific ASs in their architectures. Developers often mentioned that there were just too many ASs, but they did not know the specific types of these ASs due to the lack of definitions of various ASs. Some developers considered AS refactoring difficult because the architects who refactor architecture are often not the original architect, therefore new architects often do not maintain conceptual integrity of an architecture and run the risk that smells avoided in the original architecture may be getting worse in a subsequent version.

TABLE VIII. DIFFICULTIES IN DETECTING AND REFACTORING ARCHITECTURE SMELLS WITH THEIR EXAMPLES AND COUNTS

Difficulty	Example	Count
Cost-benefit trade-off	*I have often experienced that long-lasting architectural problems, such as application modularity, migrating between dbs, are difficult to be refactored due to urgent demands from the business side. It is also difficult to argue from developer's view point how quantitatively such refactoring would enhance the development speed and longer-term benefits. Refactoring is a delicate and time-consuming project.*	7
Lack of approaches	*Yet, there are many dimensions to architecture that these won't measure. Scalability, performance, deployment footprint, uptime, runtime characteristics, etc. How can I measure the quality of my software architecture?*	5
Lack of definitions	*I think one problem is that there is no definition for 'robust architecture'. There is only something that might be appropriate for what you're trying to do.*	5
Lack of tools	*So, for example, PMD could tell you how many times your code violates certain rules, but it cannot tell you how much work it would be to fix them, or how much extra work in the future would be incurred because you didn't fix it now.*	3
Member fluctuation	*Refactoring is often difficult because the refactorer often isn't the same person as the original designer. Therefore, he or she doesn't have the same background in the system and the decisions that went behind the original design.*	3

V. DISCUSSION

In this section, we further discuss and interpret the study results for each RQ.

Description of ASs: The results of RQ1 show that most developers used some negative words like "*bad*", "*wrong*", "*brittle*", and "*messy*" architecture to describe their architectures with smells (see TABLE IV). They tended to ask whether an architecture has smells or an architecture is bad without specifying concrete names of ASs. Some other developers used more detailed words to express smells that exist in architecture, such as violation of architecture patterns, architecture antipatterns, and layer or component problems. But developers seldom used concrete types of ASs defined in literature [10][33][35] to refer to smells in their architectures, such as cyclical dependency, scattered parasitic functionality, and ambiguous interfaces. Among these specific types of ASs, the most mentioned type of smells is cyclical dependency [33] that two or more components depend on each other.

Causes of ASs: The results of RQ2 demonstrate that three types of causes can be accounted for ASs. By analyzing the answers of the posts about why architecture has smells or architecture is bad, we identified three major causes: developers misused antipatterns in architectures, violated architecture patterns, or violated architecture design principles. But sometimes developers may have different opinions about when an architecture design is considered as an AS in various contexts. For example, when one asked whether it led to an AS by using Singleton or Service Locator in architecture, responders had a conflicting understanding. Moreover, violation of certain design principles can cause ASs, e.g., tight integration between swing UI and the business logic can cause an AS by violating the low coupling principle. In this case, design smells are intertwined with ASs.

Approaches and tools for detecting and refactoring ASs: From TABLE VI in Section IV, we conclude that most approaches for detecting and refactoring ASs are proposed to corresponding causes and aim for specific architecture problems. For example, if an ASs is caused by violating the layered architecture pattern, developers recommend a corrective action like clear separation between layers. But still, some general approaches are suggested by developers to detect and refactor ASs. For example, peer review and inspection are required to reassess your architecture.

As mentioned in some studies, ASs can be detected or refactored with dedicated tools such as Arcan [20][33], Hotspot Detector [35], and SCOOP [36], but none of these tools (as discussed in the results of RQ5) have been employed by developers, which indicates that there is a gap and mismatch between academia and industry. In addition, the tools for detecting and refactoring ASs discussed by developers are common code smell detection tools (e.g., PMD), which are not dedicated and effective to detect and refactor specific ASs.

Quality attributes affected by ASs: The most frequently concerned quality attributes when discussing ASs are Maintainability and Performance by stating that ASs would cause maintenance and performance pain spots in the long run. It is not sufficient to only talk about that ASs have detrimental

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

impacts on maintainability and performance in general without specifically discussing the detailed impact on these two quality attributes as well as on other quality attributes, e.g., understandability and modifiability. When refactoring ASs, developers need to be careful when making a tradeoff between various quality attributes affected by ASs (i.e., technical debt [38]) based on the contexts of their projects.

Difficulties in detecting and refactoring ASs: The results of RQ6 reveal that the most important challenge that impedes developers from detecting and refactoring ASs is to quantify and evaluate the cost and benefit. It is hard to persuade companies to refactor ASs without providing them the information to make a cost-benefit trade-off. Lack of tools and approaches is another difficulty that hampers developers from refactoring architecture. Moreover, unfamiliarity with ASs defined in the literature makes developers incapable to use dedicated tools in the literature to detect and refactor these ASs.

VI. Threats to Validity

In this section, we discuss three threats to the validity of this study according to the categorization [37]. Internal validity is not considered here, since this study does not address any causal relationships between variables and results.

A. Construct Validity

Construct validity concerns the extent to which an experiment measures what it is supposed to measure. In this study, construct validity refers to whether or not the methodology used can rationally support the results. The first threat to this validity is related to the search terms we used to collect ASs related posts. There is a possibility that we may leave out other terms that developers used to describe ASs. To reduce this threat, we reviewed papers related to ASs and checked various terms that are synonymous with ASs. Another threat lies in the manual analysis of the selected posts because manually coding and analyzing may introduce personal oversight and bias. To overcome this threat, we used an interpretive research method - Grounded Theory to conduct quantitative analysis. Moreover, before the formal coding, we conducted a pilot of data filtering, extraction, and analysis by the first two authors, and any conflicting results of these three activities were discussed and resolved to eliminate personal bias. After the first author finished analyzing data, the second author checked the results to verify and refine the results. Lastly, we may miss some relevant posts that refer to ASs but only use the terms e.g., "*design smell*" or "*design antipattern*", because developers hardly distinguished between architecture antipatterns and design antipatterns, as well as architecture smells and design smells as the results presented in Section IV.B. As an exploratory study, our intention is to understand ASs discussed in Stack Overflow instead of retrieving all the AS posts, and we decided not to use the search terms "*design smell*" and "*design antipattern*" due to many irrelevant results.

B. External Validity

External validity refers to the extent of generalizability of the study results. In this work, we collected data from Stack Overflow which is the largest and most popular Q&A community for developers. But we just collected and analyzed the top 50 related posts retrieved by each search term, which may result in omitting data of the remaining posts. Moreover, ASs related data in other development communities like GitHub are needed to supplement our study results, which is considered as our future work.

C. Reliability

Reliability concerns the repeatability of this study. To mitigate the threats to reliability, we specified the data collection, filtering, extraction, and analysis process in a research protocol which was discussed and confirmed by all the authors. Furthermore, a pilot study was conducted and the analysis results were checked to eliminate the misinterpretation of the results. Hence, we are confident that the similar results can be achieved when other researchers repeat this study.

VII. Conclusions and Future Work

We have performed an empirical study of developers' perception of ASs by collecting and analyzing 207 posts from Stack Overflow. The results reveal that the level of practical knowledge about ASs including the descriptions of ASs, the causes of ASs, how developers detect and refactor ASs with approaches and tools, the quality attributes affected by ASs, and the difficulties of detecting and refactoring ASs. We employed Grounded Theory to analyze the qualitative data and the main findings of this study are the following: (1) Developers often describe ASs with some general terms like "*bad*", "*wrong*", "*brittle*" or violation of architecture patterns. They hardly use the terms of specific architecture smells defined in literature. (2) The reasons why developers introduce smells in their architectures are violating architecture patterns, misusing architecture antipatterns, or violating design principles. (3) Most developers propose approaches for detecting and refactoring ASs based on the causes of these architecture smells with their specific contexts. (4) There is a lack of dedicated tools for detecting and refactoring ASs as discussed by developers, but they tend to employ code static analysis tools for ASs. Dedicated tools for ASs discussed in literature have not been adopted by practitioners. (5) Developers mainly concern about maintainability and performance of the systems affected by ASs. (6) Developers have difficulties in detecting and refactoring ASs mainly because they lack approaches and tools and cannot quantify the cost.

The study results provide several implications for our future research including: (1) Employing semi-automatic approaches to extract and analyze AS related data from all the related posts. (2) Exploring the reliability of the results about ASs in other developer communities like GitHub. (3) Conducting other empirical studies such as surveys to supplement our results. (4) Proposing a catalogue of ASs and refactoring approaches and tools by synthesizing relevant studies.

References

[1] Easterbrook, S., Singer, J., Storey, M.A. and Damian, D., 2008. Selecting Empirical Methods for Software Engineering Research. Guide to Advanced Empirical Software Engineering, Chapter 11, Springer, pp. 285-311.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.

[2] Samarthyam, G., Suryanarayana, G. and Sharma, T., 2016. Refactoring for software Architecture smells. In: Proceedings of the 1st International Workshop on Software Refactoring (IWoR). Singapore, Singapore, pp. 1-4.

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., 1999. Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman.

[4] Yamashita, A. and Moonen, L., 2013. Do developers care about code smells? An exploratory survey. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE). Koblenz, Germany, pp. 242-251

[5] Suryanarayana, G., Samarthyam, G. and Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann Publishers Inc.

[6] Brown, W.J., Malveau, R.C., Mowbray, T.J. and Wiley, J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons.

[7] Moha, N., Gueheneuc, Y.G. and Duchien, A.F., 2010. Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1), pp. 20-36.

[8] Garcia, J., Popescu, D., Edwards, G. and Medvidovic, N., 2009. Toward a Catalogue of Architectural Bad Smell. In: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems (QoSA). East Stroudsburg, PA, USA, pp. 146-162.

[9] Grant, S. and Betts, B., 2013. Encouraging user behaviour with achievements: an empirical study. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR). San Francisco, CA, USA, pp. 65-68.

[10] Le, D., Carrillo, C., Capilla, R. and Medvidovic, N., 2016. Relating architectural decay and sustainability of software systems. In: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). Venice, Italy, pp. 178-181.

[11] Vidal, S., Vazquez, H., Diaz-Pace, J.A., Marcos, C., Garcia, A. and Oizumi, W., 2015. JSpIRIT: a flexible tool for the analysis of code smells. In: Proceedings of the 34th International Conference of Chilean Computer Science Society (SCCC), Santiago, Chile, pp. 1-6.

[12] PMD, 2018. PMD Source Code Analyzer. https://pmd.github.io/

[13] Tsantalis, N., Chaikalis, T. and Chatzigeorgiou, A., 2008. JDeodorant: Identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), Athens, Greece, pp. 329-331.

[14] Gupta, A., Suri, B. and Misra, S., 2017. A systematic literature review: Code bad smells in Java source code. In: Proceedings of the 17th International Conference on Computational Science and Its Applications (ICCSA), Trieste, Italy, pp. 665-682.

[15] Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D. and Zhang, D., 2003. Using multiple views to model and analyze software architecture: An experience report. Technical Report USC-CSE-2003-508, University of Southern California.

[16] Bertran, I.M., 2011. Detecting architecturally-relevant code smells in evolving software systems. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, pp. 1090-1093.

[17] Lippert, M. and Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. John Wiley & Sons.

[18] Amirat, A., Bouchouk, A., Yeslem, M.O. and Gasmallah, N., 2012. Refactor Software architecture using graph transformation approach. In: Proceedings of the 2nd International Conference on Innovative Computing Technology (INTECH), Casablanca, Morocco, pp. 117-122.

[19] Kaur, M. and Kumar, P., 2014. Spotting the phenomenon of bad smells in MobileMedia product line architecture. In: Proceedings of the 7th International Conference on Contemporary Computing (IC3), Noida, India, pp. 357-363.

[20] Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M. and Nitto, E. D., 2017. Arcan: A Tool for Architectural Smells Detection. In: Proceedings of IEEE International Conference on Software Architecture Workshops (ICSAW). Gothenburg, Sweden, pp. 282-285.

[21] Soliman, M., Galster, M., Salama, A.R. and Riebisch, M., 2016. Architectural knowledge for technology decisions in developer communities: An exploratory study with stackoverflow. In: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy, pp. 128-133.

[22] Meldrum, S., Licorish, S.A. and Savarimuthu, B.T.R., 2017. Crowdsourced Knowledge on Stack Overflow: A Systematic Mapping Study. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE), Karlskrona, Sweden, pp. 180-185.

[23] Soliman, M., Salama, A.R., Galster, M., Zimmermann, O. and Riebisch, M., 2018. Improving the Search for Architecture Knowledge in Online Developer Communities. In: Proceedings of the 15th IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, pp. 186-195.

[24] Bi, T., Liang, P. and Tang, A., 2018. Architecture Patterns, Quality Attributes, and Design Contexts: How Developers Design with Them?. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, pp. 49-58.

[25] Zou, J., Xu, L., Yang, M., Zhang, X. and Yang, D., 2017. Towards comprehending the non-functional requirements through Developers' eyes: An exploration of Stack Overflow using topic analysis. Information and Software Technology, 84, pp. 19-32.

[26] Venkatesh, P.K., Wang, S., Zhang, F., Zou, Y. and Hassan, A., 2016. What concerns do client developers have when using web APIs? An empirical study of developer forums and stack overflow. In: Proceedings of the 23rd IEEE international conference on web services (ICWS), San Francisco, CA, USA, pp. 131-138.

[27] Linares-Vásquez, M., Dit, B. and Poshyvanyk, D., 2013. An exploratory analysis of mobile development issues using stack overflow. In: Proceedings of 10th IEEE Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, pp. 93-96.

[28] Yang, X.L., Lo, D., Xia, X., Wan, Z.Y. and Sun, J.L., 2016. What security questions do developers ask? a large-scale study of stack overflow posts. Journal of Computer Science and Technology, 31(5), pp. 910-924.

[29] Tahir, A., Yamashita, A., Licorish, S., Dietrich, J. and Counsell, S., 2018. Can you tell me if it smells?: A study on how developers discuss code smells and anti-patterns in Stack Overflow. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE), Christchurch, New Zealand, pp. 68-78.

[30] Abdalkareem, R., Shihab, E. and Rilling, J., 2017. What do developers use the crowd for? a study using stack overflow. IEEE Software, 34(2), pp. 53-60.

[31] Corbin, J. and Strauss, A., 2007. Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, 3rd edition Sage Publications.

[32] Glaser, B.G. and Strauss, A.L., 2009. The Discovery of Grounded Theory: Strategies for Qualitative Research. Transaction Publishers.

[33] Fontana, F.A., Pigazzini, I., Roveda, R. and Zanoni, M., 2016. Automatic detection of instability architectural smells. In: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME). Raleigh, NC, USA, pp. 433-437.

[34] Rizzi, L., Fontana, F.A. and Roveda, R., 2018. Support for architectural smell refactoring. In: Proceedings of the 2nd International Workshop on Refactoring (IWoR), Montpellier, France, pp. 7-10.

[35] Mo, R., Cai, Y., Kazman, R. and Xiao, L., 2015. Hotspot patterns: The formal definition and automatic detection of Architecture Smells. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, QC, Canada, pp. 51-60.

[36] Macia, I., Arcoverde, R., Cirilo, E., Garcia, A. and von Staa, A., 2012. Supporting the identification of architecturally-relevant code anomalies. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), pp. 662-665.

[37] Brewer, M.B. and Crano, W.D., 2014. Research Design and Issues of Validity. Handbook of Research Methods in Social and Personality Psychology, Second Edition, pp. 11-26.

[38] Li, Z., Avgeriou, P. and Liang, P., 2015. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101, pp. 193-220.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on June 29,2023 at 21:01:59 UTC from IEEE Xplore. Restrictions apply.