The Perception of Architectural Smells in Industrial Practice

Darius Sas, University of Groningen

Ilaria Pigazzini, University of Milano–Bicocca

Paris Avgeriou, University of Groningen

Francesca Arcelli Fontana, University of Milano–Bicocca

// Architectural technical debt (TD) is the most significant type of TD in industrial practice. Our goals in this study were to better understand the phenomenon of architectural smells, help practitioners better manage them, and offer relevant support to researchers. //

THE METAPHOR OF technical debt (TD) reflects the technical compromises that software practitioners make to achieve a short-term advantage at the expense of creating a technical context that increases complexity and cost in the long-term.[1] TD can be incurred throughout the entire software development process, so multiple types can be identified (e.g., requirements, architecture, and code).[2] Architectural TD (ATD) was found to be one of the most significant types of TD, as, typically, key architectural decisions are made very early in the software lifecycle and, thus, have a stronger impact.[3]

Architectural smells (AS) are one type of ATD: all AS instances are ATD items, but not all ATD items are AS.[4] *AS* are defined as "commonly (although not always intentionally) used architectural decisions that negatively impact system quality."[5]

AS manifest themselves in the system as undesired dependencies, an unbalanced distribution of responsibilities, excessive coupling between components, and many other forms that break one or more software design principles and good practices, ultimately affecting maintainability and evolvability.[6] We note that the presence of AS does not always inevitably indicate that there is a problem, but it points to places in the system's architecture that should be further analyzed.[6]

Despite the recent attention from the research community on the topic,[4] few studies investigated how practitioners understand AS and experience the associated maintainability issues in the real world.[7] To address this shortcoming, we interviewed 21 software developers and architects to collect their opinions and experiences from industrial practice regarding three research questions (RQs) (see "Study Design").

Specifically, we focus on how practitioners perceive AS, what maintenance and evolution issues they associate with AS, and how they introduce and deal with them in terms of adopted practices and tools. The goal is to enrich researchers' understanding of AS and inform practitioners on how they manifest themselves

Digital Object Identifier 10.1109/MS.2021.3103664
Date of current version: 22 October 2021

STUDY DESIGN

We performed a case study to collect experiences from industry regarding three RQs:

- *RQ1*: How are AS perceived by practitioners?
- *RQ2*: What are the maintainability and evolvability issues experienced by practitioners that relate to the presence of AS in the system?
- *RQ3*: How do practitioners introduce and deal with AS?

For practitioners, answering these questions can help them understand and relate to issues experienced by others, obtain deeper knowledge about AS, and learn how to manage them. Researchers, on the other hand, can better understand the real-world problems experienced by practitioners and how exactly AS contribute to TD.

We collected data by interviewing 21 practitioners from three companies in Europe operating in two different domains (Embedded Systems and Enterprise Applications Development) with three main programming languages (C, C++, and Java). The first company provided 12 participants; the second, six; and the third, three. Practitioners' backgrounds vary from a few years of activity (junior developers) up to 25 years of practice (architects). The average size of their projects is about 50 million lines of code (LOC) for the first company, from 500,000 to 1,000,000 LOC for the second, and from 250,000 to 750,000 LOC for the third.

Interviews were semistructured, and each lasted approximately 30 min. We chose to use interviews because they allow for follow-up questions and clarifications, ensuring that participants have understood the questions. Further details about the design of this study can be found in the online appendix (available in https://doi.org/10.1109MS.2021.3103664).

in a real-world scenario, ultimately supporting better AS management.

While there exist several kinds of AS, we limited our scope to the four types that are detected by most of the available tools[8] and are among the most important currently described in the literature[7]:

- Cyclic dependency (CD) is a set of software artifacts (e.g., classes, files, packages, components, and so on) that depend upon each other, thus creating a cycle. CD breaks the acyclic dependencies principle[9] and increases coupling.
- Hub-like dependency (HL) is an artifact that has an excessive number of incoming and outgoing dependencies, thus creating a hub. HL breaks the modularity of the system as the hub is overloaded with responsibilities and exacerbates the dependency structure of the system.
- Unstable dependency (UD) is a package (or any similar construct—e.g., a component) that has too many dependencies to packages that are less stable than itself, thus increasing its reasons to change. A package is said to be stable if it is resilient to changes in neighboring packages. UD breaks the stable dependency principle ("Depend in the direction of stability"[9]) because the affected package depends on packages less stable than itself.
- A God component (GC) is a package (or component) whose size [measured using lines of code] is noticeably bigger than the other components in the system.[6] A GC breaks system modularity and aggregates too many concerns into a single package.

It is important to note that participants in our study were asked not to limit themselves to these four smells only and were free to mention experiences related to different types of AS.

Results

How AS Are Perceived (RQ1)

Participants reported being the most familiar with GCs among the four studied AS; several practitioners reported personal experiences in managing this kind of smell. GCs are perceived as a common cause of maintenance issues as well as reduced evolvability of the affected component, mainly as a result of the high level of complexity that characterizes its instances.

In particular, almost all practitioners, except for two architects, had rather strong opinions on this AS and underlined its importance vividly. The two architects, instead, expressed some skepticism when discussing its importance and disregarded it, as they saw no added technical value in splitting a GC.

Opinions on CD were generally aligned, and most interviewees considered it to be detrimental for maintainability, reliability, and testability. Concerns about reliability (e.g., deadlocks) were mostly expressed by participants working on C/C++ projects, highlighting that, even if some

CD instances have not caused issues yet, they pose a high risk for future undertakings.

On the other hand, interviewees working with Java perceived CD as less detrimental than other smell types, like GC. This difference in perception is probably due to the different application domains of the companies and not only the differences between Java and C/C++.

We note that, typically, AS are the symptom of a bigger and more profound issue in the architecture[6] that needs to be studied case by case. However, in cases where CD affected reliability and testability, its very presence was considered as the problem that developers were trying to resolve.

Opinions were much more polarized when the HL smell was discussed. Some participants mentioned that

- it should not be considered a problem because it could be the result of an intentional design decision
- it should not be a cause of concern as long as it is understandable
- it is easy to solve, as one participant expressed.
- However, other respondents (and especially the ones working with Java) mentioned that HL is very important to avoid because it is not easy to manage and hinders both maintainability and evolvability by making it hard to understand how to insert new code in the presence of HL.

Concerning UD, participants generally perceived it as a threat to both maintainability and evolvability, highlighting their concerns about the change ripple effects associated with it and underlining the importance of avoiding dependencies toward packages that constantly evolve. Nevertheless, one developer expressed doubts about the importance of this AS, while a few more stated that they did not fully understand it and gave no feedback about it.

From these results, it appears that, while all AS are considered detrimental, they are perceived differently by practitioners depending on their past experiences, educational background, and application domain: GC and CD are perceived as the most important ones, HL is considered "manageable," and UD is seen as detrimental but not critical. It is also important to take into account that UD is less visible than the other smells: one cannot tell by looking at a package that it is less stable than another one without employing dedicated tooling.

Finally, we observed the existence of a slight correlation between the experience of interviewees and type of concerns expressed about an AS. Junior participants tended to be more concerned about short-term problems (e.g., the presence of CDs and their impact on the deployed system), while senior interviewees were keener on long-term evolvability and team-related matters (e.g., new team members making changes to a GC).

How AS Impact Maintenance and Evolution (RQ2)

The participants discussed plenty of anecdotes and experiences about maintenance and evolution issues that they associated with the presence of AS. Almost all anecdotes about GCs involve the difficulty of understanding the functionality provided by the component, mainly caused by the excessive internal entanglement of files (or classes), significant amount of functionality implemented, and the way functionality is scattered across the component.

The relationship between GCs and code duplications was also frequently discussed. Components affected by a GC do not provide fine-grained classes that can be easily reused inside or outside the component but, rather, large and entangled classes. Hence, when developers need to reuse an existing functionality, they prefer to copy the entire class and adapt it for the new purpose instead of extracting a small, reusable functionality. On top of creating duplicated code, this also further enlarges the existing GC.

The experiences about CD are rather diverse and range from dealing with deadlocks and low throughput to an unclear chain of command among components and poor separation of concerns in general. Cycles were also reported as an "intertwined mess" that is hard to understand—e.g., when there is a package that requests data from another package, which, in turn, requests it back from the initial package.

These issues required a significant amount of effort to fix or deal with them along the way, and, in some cases, they showed up only in production or at the customer level. Participants also mentioned problems that had a more widespread impact; for example, a cycle prevented the creation of a microservice out of a subset of packages, as all of the packages in the cycle had to be included in the microservice. (The desired functionality could not be isolated.)

Concerning HL, practitioners associated it with two types of issues:

- difficulty of understanding the logic in the central component
- change ripple effects propagating from the components that the central component depends upon to the components depending on it, mentioning also a possible overlap with UD.

The former was usually associated with how the central component exposes its functionality through its interface. The latter caused changes to unexpected parts of the system that practitioners did not expect to relate to the initial change, during activities such as bug fixing.

The maintenance issues most associated with UD were change ripple effects. In several instances, practitioners reported that functional changes to a certain component (or package) also required several files in other components to change as well. As reported by two participants, the possibility of changes propagating to other components increases the difficulty of making changes: practitioners are forced to only make changes compatible with the other components to avoid changing and recompiling those other components.

How AS Are Introduced and Managed (RQ3)

Participants reported their experiences with how they get to introduce an AS in the system. Some interviewees admitted that it often happens by design; for instance, concerning GCs, the component or the file is intended to be large. Subsequently, as reported by other interviewees, developers tend to underestimate the severity of the introduced GC, while the incremental changes applied to it contribute to making it even larger.

In other cases, AS are introduced inadvertently. For example, participants reported that a bad separation of concerns at design time or the wrong exploitation of class inheritance can result in CD. Another respondent mentioned that they used to create a dedicated interface to hide unstable components behind it as a "practice" to avoid the propagation of changes; however, this is precisely the description of a UD smell, which is being misinterpreted as a good practice.

In many cases, introducing AS seems unavoidable and accepted as a "necessary evil." For example, one participant explained that, in view of an imminent deadline, they focus on developing the new feature and having a first structure of the code without caring about its maintainability.

Moving on to the management of AS, we asked participants about their experiences with AS refactoring. Most of them had experience with the refactoring of GCs, in particular, the practice of splitting the component into smaller pieces by applying incremental changes or detaching the smallest, easiest subcomponents first.

One interviewee had managed to break a case of CD by remodeling the involved dependencies to follow a hierarchical structure; others reported creating replacement interfaces and slowly migrating clients to them while refactoring the existing components. In contrast, developers do not commonly refactor HL because of the required effort; if they can, they tend to "code around it" without removing it when developing new features, allowing it to persist. One interesting reason mentioned for not refactoring an AS is the absence of a comprehensive regression test suite.

Concerning practices that support the refactoring of AS, some participants mentioned the usage of SonarQube to keep the code readable and maintainable; this can ease the refactoring of AS since, often, the poor quality of the code makes refactoring even more difficult and time consuming. Another respondent indicated pair programming and the help of senior developers as valid support.

However, not all of the interviewees reported the adoption of refactoring practices. Some even pointed out that they avoid refactoring because their clients do not pay for refactoring time, and, as long as the system has no visible problems in production, they do not intervene.

Finally, we also asked whether practitioners use tools to manage AS. SonarQube was mentioned by quite a few respondents, but only once regarding an AS (i.e., to detect cycles). Besides that, practitioners do not rely on any specific tool to manage AS. Nonetheless, participants did mention features that they would like to have in an ideal tool that manages AS. Due to space limitations, the features are reported in our online appendix (available in https://doi.org/10.1109MS.2021.3103664), and we created a mind map to summarize the results of all three RQs in Figure 1.

Discussion and Implications

The presented results indicate that AS clearly help incur ATD: they have a direct, architecture-level impact on the maintainability and evolvability of the affected parts. AS make changes harder to implement by increasing the effort required to understand the implications of a change, making it easy to underestimate the effort necessary for the change and hard to plan ahead.

Practitioners are aware and well informed about good design practices, but they struggle to follow them diligently, often prioritizing delivering a feature over good design. Fowler calls this reckless and deliberate TD,[10] because practitioners understand the long-term implications of their decisions but still decide to incur TD. By doing so, practitioners are forced, sooner rather than later, to apply refactorings before

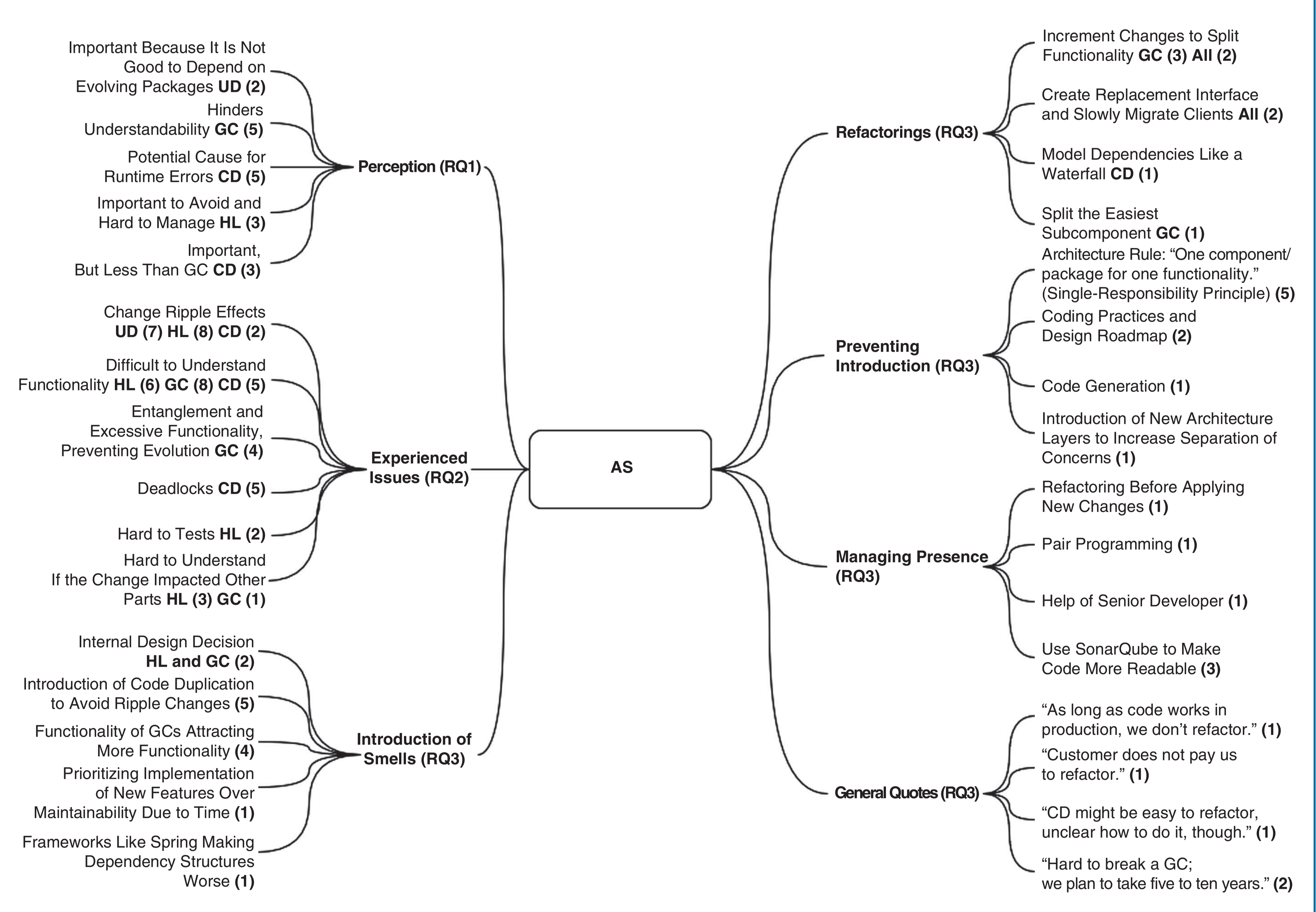
AS
Perception (RQ1)
Important Because It Is Not Good to Depend on Evolving Packages UD (2)
Hinders Understandability GC (5)
Potential Cause for Runtime Errors CD (5)
Important to Avoid and Hard to Manage HL (3)
Important, But Less Than GC CD (3)
Experienced Issues (RQ2)
Change Ripple Effects UD (7) HL (8) CD (2)
Difficult to Understand Functionality HL (6) GC (8) CD (5)
Entanglement and Excessive Functionality, Preventing Evolution GC (4)
Deadlocks CD (5)
Hard to Tests HL (2)
Hard to Understand If the Change Impacted Other Parts HL (3) GC (1)
Introduction of Smells (RQ3)
Internal Design Decision HL and GC (2)
Introduction of Code Duplication to Avoid Ripple Changes (5)
Functionality of GCs Attracting More Functionality (4)
Prioritizing Implementation of New Features Over Maintainability Due to Time (1)
Frameworks Like Spring Making Dependency Structures Worse (1)
Refactorings (RQ3)
Increment Changes to Split Functionality GC (3) All (2)
Create Replacement Interface and Slowly Migrate Clients All (2)
Model Dependencies Like a Waterfall CD (1)
Split the Easiest Subcomponent GC (1)
Preventing Introduction (RQ3)
Architecture Rule: "One component/ package for one functionality." (Single-Responsibility Principle) (5)
Coding Practices and Design Roadmap (2)
Code Generation (1)
Introduction of New Architecture Layers to Increase Separation of Concerns (1)
Managing Presence (RQ3)
Refactoring Before Applying New Changes (1)
Pair Programming (1)
Help of Senior Developer (1)
Use SonarQube to Make Code More Readable (3)
General Quotes (RQ3)
"As long as code works in production, we don't refactor." (1)
"Customer does not pay us to refactor." (1)
"CD might be easy to refactor, unclear how to do it, though." (1)
"Hard to break a GC; we plan to take five to ten years." (2)

FIGURE 1. A mind map summarizing the perception, experiences, prevention, introduction, and presence of AS as described by participants. In parentheses, we report the number of data points and, if appropriate, the type of associated AS.

proceeding with the implementation of new features (as mentioned by the participants) and pay a considerable amount of TD interest every time they need to extend the system.

As emerged from the interviews, TD is also incurred inadvertently,[10] either recklessly because of poor knowledge about the design of the parts affected by the change (e.g., a component requesting a parameter that belongs to itself from another component), or prudently because the optimal design solution becomes clear only after implementing the chosen solution. The introduction of TD through nonoptimal solutions that is then detected as an AS is not automatically controlled, as we observed a lack of adoption of tooling dedicated to managing AS—practitioners mostly focus on code TD.

At any rate, regardless of the how, incurring TD is inevitable and inherent to the software development process, so practitioners must adopt practices that enable its management. Similar to any other type of TD item, the first step in managing AS is detecting them. Azadi et al.[8] recently provided a list of tools that detect AS for practitioners to consider.

Another—even more important—step is prevention. Practitioners should pay particular attention to how they create internal dependencies, as there is a fine balance between changeability and the number of dependencies per file: too many, and files become entangled, making the system hard to modify and giving rise to GCs and CD; too few, and the system is also hard to modify because fewer classes are reused (a treelike dependency graph[6]), resulting in multiple classes implementing similar functionality (and applying the same change to all of them is repetitive).

Practitioners should carefully balance how these dependencies are created by devising clear architectural rules that prevent the creation of undesired dependencies that end up generating AS. Our research work to date has focused precisely on addressing these issues, culminating in the development of Arcan, a tool to make AS detection and dependency analysis as easy as possible to practitioners who work either in Java or C/C++—and, soon, Python and C#.

ABOUT THE AUTHORS

DARIUS SAS is a Ph.D. student at the Bernoulli Institute for Mathematics, Computer Science, and Artificial Intelligence, University of Groningen, Groningen, 9747 AG, The Netherlands. His research interests include architectural technical debt elimination in embedded systems and the interplay among technical debt, energy efficiency, and dependability. Sas received his master's degree in computer science from the University of Milano–Bicocca, Milan, Italy. Contact him at d.d.sas@rug.nl.

ILARIA PIGAZZINI is a Ph.D. student in computer science at the Department of Computer Science, Systems, and Communications, University of Milano–Bicocca, Milan, 20216, Italy. Her research interests include reverse engineering, architectural smell detection, and the refactoring of object-oriented systems. Pigazzini received her M.Sc. in computer science from the University of Milano–Bicocca. Contact her at i.pigazzini@campus.unimib.it.

PARIS AVGERIOU is a professor of software engineering at the University of Groningen, Groningen, 9747 AG, The Netherlands. His research interests include software architecture with a strong emphasis on architecture modeling, knowledge, evolution, analytics, and technical debt. Avgeriou received his Ph.D. from the Department of Electrical and Computer Engineering of the National Technical University of Athens. Contact him at p.avgeriou@rug.nl.

FRANCESCA ARCELLI FONTANA is a full professor and head of the Software Evolution and Reverse Engineering Lab at the University of Milano–Bicocca, Milan, 20216, Italy. Her research interests include software evolution, reverse engineering, managing technical debt, and software quality assessment. Fontana received her Ph.D. in computer science from the University of Milano. Contact her at francesca.arcelli@unimib.it.

Acknowledgments

This article has supplementary downloadable material available at https://doi.org/10.1109MS.2021.3103664 provided by the authors.

References

1. P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl Seminar 16162)," *Dagstuhl Rep.*, vol. 6, no. 4, pp. 110–138, 2016. 10.4230/DagRep.6.4.110

2. N. S. R. Alves, T. S. Mendes, M. G. De Mendonça, R. O. Spinola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Nov. 2016. doi: 10.1016/j.infsof.2015.10.008.
3. N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? software practitioners and technical debt," in *Proc. 10th Joint Meeting on Foundat. Softw. Eng. – ESEC/FSE 2015*, 2015, pp. 50–60. doi: 10.1145/2786805.2786848.
4. R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: The research landscape," in *Proc. ACM/IEEE Int. Conf. Tech. Debt.*, 2018, pp. 11–20. doi: 10.1145/3194164.3194176.
5. J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 255–258, doi: 10.1109/CSMR.2009.59.
6. S. R. Lippert, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Hoboken, NJ: Wiley, 2006.
7. F. Arcelli Fontana, F. Locatelli, I. Pigazzini, and P. Mereghetti, "An architectural smell evaluation in an industrial context," in *Proc. Int. Conf. Softw. Eng. Adv.*, 2020, vol. 78.
8. U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: A catalogue proposal," in *Proc. Int. Conf. Tech. Debt (TechDebt 2019)*, 2019, pp. 88–97. doi: 10.1109/TechDebt.2019.00027.
9. R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Englewood Cliffs, NJ: Prentice-Hall.
10. M. Fowler, "Technical Debt Quadrant," 2009. Accessed: Apr. 4, 2021. [Online]. Available: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Digital Object Identifier 10.1109/MS.2021.3115640