On Developing and Improving Tools for Architecture-Smell Tracking in Java Systems

Philipp Gnoyke
Otto-von-Guericke University, Magdeburg, and *KSB SE & Co. KGaA*, Pegnitz, Germany
philipp.gnoyke@t-online.de

Sandro Schulze
Anhalt University of Applied Sciences
Köthen, Germany
sandro.schulze@hs-anhalt.de

Jacob Krüger
Eindhoven University of Technology
Eindhoven, The Netherlands
j.kruger@tue.nl

***Abstract*—Architecture smells indicate violations of software-design principles. So, identifying and assessing architecture smells facilitates refactorings to reduce technical debt and ensure maintainability. Detecting architecture smells in only one version at a time provides a static and limited picture, since, for example, historical trends remain obfuscated. Both, for practitioners and researchers, obtaining information on how specific architecture smells evolved over time can yield valuable insights, be it for avoiding the growth of critical smells, grasping the code's degradation, or getting a better understanding of development processes. To support such analyses, we developed our tool `AsTdEA`, which tracks architecture smells throughout a system's evolution. `AsTdEA` runs a modified version of the architecture-smell detection tool `Arcan` and allows the automated batch-processing of multiple versions of one or multiple systems. First, `AsTdEA` generates data on the components that are involved in each smell on a version-to-version basis and how these *intra-version* smells are related with one another across the entire system history, forming *inter-version* smells. Second, for every intra-version smell, inter-version smell, and system version, `AsTdEA` outputs a multitude of properties, which we have already used for multiple empirical studies. In this paper, we show the implementation and use of `AsTdEA`, as well as the lessons that we learned during its development and how we want to improve it in the future.**

***Index Terms*—Architecture Smells, Software Quality, Tracking, Software Tools, Source-Code Analysis, Software Evolution**

I. Introduction

To remain relevant, software must be continuously maintained and evolved according to ever-changing requirements. If a thorough quality assurance is not in place and if updates are implemented without adhering to software-design principles or to the intended architecture, software systems start to exhibit *smells* that cause quality degradation over time (a.k.a. aging or decaying) [1], [7], [23]. Software degradation reduces maintainability, increases the costs for new features, and can easily cause more bugs, which is commonly described with the metaphor of technical debt [8], [9], [15], [37]. To identify the violations of software-design principles, software developers can use smell detection tools [3], [6]. As one particular type of smells, Architecture Smells (ASs) represent violations against best practices for designing software architectures, and thus identifying them can be an essential means for assuring a system's quality [11], [18]. However, detecting ASs in only the latest version of a software system provides limited insights, for example, because developers cannot see what ASs are growing rapidly or from what parts of a system an AS originated. Gaining insights into such properties is helpful for practitioners developing a specific system and for researchers understanding general patterns on software evolution. Consequently, AS tracking tools are needed to provide support for analyzing and improving the quality of a software system [27].

In this paper, we present our open-source AS tracking tool `AsTdEA`, which is available in an open-access repository.[1] To develop `AsTdEA`, we modified `Arcan`, an academic tool for AS detection, that detects three common types of ASs in Java systems: cyclic, hub-like, and unstable dependencies [3]. `AsTdEA` can analyze the entire history of one or multiple Java systems in a batch-mode by running `Arcan`, generating script-readable output for further processes like visualizations or statistical tests. We implemented novel concepts for analyzing the evolution of ASs within `AsTdEA` and employed it in empirical studies [12]. For instance, by linking related intra- to inter-version smells (cf. Section III-A) and calculating their properties, `AsTdEA` provides data that would not be available by simply running `Arcan` sequentially on several versions.

In the remainder of this paper, we first define fundamental terms in Section II before highlighting the benefits and use of `AsTdEA` in Section III. Then, we provide an overview of our implementation in Section IV. Moreover, we present a series of overcome and persisting challenges in Section V, which serve as lessons learned for other tool developers and as a basis for future research in the direction of ASs. In Section VI, we sketch our current ideas for further enhancing `AsTdEA` that we aim to address in future work. Finally, we compare `AsTdEA` with the related tool `ASTracker` in Section VII and conclude this paper in Section VIII.

II. Architecture Smells

A smell is an indicator for a deeper problem within a software system [10], such as violated design principles or rules. Such principles and rules, in turn, represent best practices for ensuring maintainability when structuring source code [18]. Originally, the notion of smells stems from code smells [10], which represent structures within the code that may hint at a quality or maintainability problem. In contrast, a smell that impacts a system's architecture, specifically how the system is structured into components and how these interact or depend

[1] https://github.com/PhilippGnoyke/AsTdEA

on each other [34], is referred to as an *architecture smell (AS)* [11], [18]. In the context of Java systems, a component can either represent a class or a whole package. ASs often reduce the maintainability of software by increasing coupling and by propagating changes in one component to otherwise unaffected components, called ripple effects [13], [20]. Smells can be detected with automated tooling, such as the academic tool `Arcan` for ASs [3], [6]. `Arcan` detects three types of ASs in Java systems, which we also focus on with our tool `AsTdEA` and research. We shortly outline the definition and properties of each of these types of ASs in the following.

First, a *cyclic dependency* (CD) constitutes two or more components that mutually depend on each other, which increases the risk of ripple effects [18], [32]. We consider both CDs among classes (i.e., *class-level*) and among packages (i.e., *package-level*). CDs can be defined as simple cyclic paths with no repeated vertices (i.e., components) or as strongly connected components with any number of edges between a set of vertices (cf. Section III-A) [21]. Second, a *hub-like dependency* (HD) represents a hub class with a large number of both incoming and outgoing dependencies. This results in a bottleneck in the dependency structure and the propagation of changes through the hub [4], [32]. Lastly, an *unstable dependency* (UD) is a relatively stable component that depends on less stable components. Stability refers to the rate of change a component has exhibited or is expected to exhibit [4], [22]. `Arcan` employs Martin's instability metric to detect UDs, which assumes that components with many dependers are less likely to change [19]. We are focusing on package-level UDs.

A concept related to smells is technical debt, which is a metaphor to summarize and communicate various quality issues in a software system akin to financial debt [5], [9], [15], [32]. For example, technical debt can be accrued to speed up development by disregarding the intended architecture, design principles and rules, documentation, or software testing. The consequently reduced maintainability leads to "interest payments" in the form of increased efforts needed during the further development, as well as the higher likelihood of bugs. Technical debt is repayed by employing refactorings, but its uncontrolled growth can lead to "technical bankruptcy."

III. AsTdEA

In this section, we highlight the benefits and use case of `AsTdEA`, starting with the novel concepts we have implemented, before showcasing the output the tool generates, and finally explaining how `AsTdEA` can be used.

A. Concepts Supported by AsTdEA

To conduct our own and partly inspired by related research [2], [12], [24], [25], [27], [28], we have incrementally introduced novel concepts for AS tracking into `AsTdEA`. Specifically, these concepts include:

Differentiating between intra-version and inter-version ASs. Intra-version ASs are AS instances in one particular version. In contrast, inter-version ASs represent a set of related intra-version ASs that span multiple versions of a system.

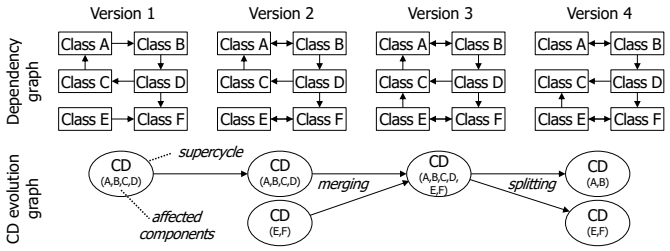

Fig. 1. From cycles in the dependency graph to CD evolution graphs.

By distinguishing the two, we have clarified the terminology and concepts we use—with our focus being on analyzing the evolution of ASs in the source code (i.e., inter-version ASs).

Differentiating between subcycles and supercycles for CDs. Subcycles are simple cyclic paths, while supercycles are potentially complex strongly connected components. Subcycles represent possible routes of change propagation and are easy to understand/visualize, while introducing the concept of supercycles facilitates accurate tracking of CDs during a system's evolution. That is because removing a single dependency between two affected components breaks a subcycle, while tangled supercycles usually require considerable changes to the code to be resolved. Specifically, we regard two intra-version CDs in adjacent versions as related if they share at least two components, since every component can at most participate in one supercycle and there exists a cyclic path between any two components in a supercycle.

Considering merging and splitting in the evolution of CDs. Adding or removing edges can change the structure of CDs. For instance, adding edges into the dependency structure of a system can cause two supercycles to merge, or to split when removing edges. Considering merging and splitting improves the tracking of evolving CDs, and allows for novel analyses. For instance, we found CDs with many affected components to often be the cause of extensive mergings over time.

Representing CD evolution as CD evolution graphs. Due to the aforementioned merging and splitting, complex CDs cannot be tracked in a one-dimensional manner. To resolve this problem, we have developed the idea of CD evolution graphs to properly document and visualize the evolution of CDs. Within a CD evolution graph, every intra-version CD is represented by a vertex, while predecessor/successor relationships between intra-version CDs are drawn as edges. We visualize a conceptual example of a CD evolution graph in Figure 1.

Tracking HDs and UDs in a two-step process. HDs and UDs are both centered around a central component (hub class and stable package depending on less stable packages). We use that component to link related intra-version ASs in adjacent versions if the same component can be found in both versions. Only in a second step, we perform a set comparison of the remaining components if, for instance, we could not match the central component due to renaming. This improved the tracking of HDs and UDs across different system versions.

B. Output

AsTdEA generates a series of .csv files that contain information on the properties and affected components of all analyzed ASs. These files can be parsed by other applications (e.g., to create visualizations), queried by scripts to analyze a particular system at hand, or used to perform empirical studies to better understand the evolution of ASs and the respective software system. For instance, an intra-version property can be analyzed as a time series if the respective AS is part of an inter-version AS.

In more detail, the output includes various intra-version AS properties, such as an AS's order (number of affected components), size (number of edges between components), centrality (PageRank value in the dependency structure), or overlap ratio (share of components shared with other ASs). Moreover, AsTdEA generates properties for inter-version ASs, system versions, and the entire system history. Particularly, we want to highlight that AsTdEA provides a technical-debt quantification based on Roveda et al. [24], [25] that assigns a severity score and technical-debt value to every AS. Furthermore, AsTdEA assigns one of the eight shapes proposed by Al-Mutawa et al. [2] to every CD.

For every intra-version AS, AsTdEA provides a list of affected components with information on their role in the respective smell (e.g., for HDs: hub class, afferent component, efferent component). Also, AsTdEA provides a dependency matrix between all affected components within each CD. One level of abstraction higher, AsTdEA generates .csv files linking inter-version ASs with the intra-version ASs that constitute them—again with additional CD-specific output for edges in CD evolution graphs.

C. Usage

AsTdEA can be used via a terminal or by running its main method in another Java program. Exact instructions are provided in our repository.[1] In the input folder, the user has to provide data for every system that shall be analyzed. First, this data includes one .jar file or a folder of .jar files per system version that the user wants to analyze. Second, AsTdEA requires metadata for the number of lines of code and the release day of each version to compute its normalized and time-based metrics. AsTdEA sorts the versions automatically according to their version numbers, unless the user provides explicit metadata for a different sorting. The input and output folder locations can be customized, but default values exist so that AsTdEA can be executed without any arguments. By default, AsTdEA runs Arcan on every provided version before tracking inter-version ASs and computing the properties of ASs. Alternatively, Arcan can be suppressed in case that its output has been generated previously, for example, as part of a recurring code analysis. This also tackles the issue that analyzing larger systems with Arcan can take a considerable amount of time (cf. Section IV-A). For the same purpose, AsTdEA provides information upon finishing the scan of every version, but avoids extensive console output that would slow down the program execution.

IV. Implementation

In this section, we provide details on the implementation of AsTdEA, our modifications of Arcan, and how scripts are integrated into the toolchain. We provide an overview of our toolchain in Figure 2.

A. Modified Arcan

To implement the concepts we introduced in Section III-A and generate the described output (cf. Section III-B), we had to modify the available open-source version of Arcan (release 1.2.1).[2] We provide our modified version of Arcan in another open-access repository.[3] Specifically, we added technical-debt quantification, changed the main definition of CDs from subcycles to supercycles, altered the CD shape classification to match the original algorithm by Al-Mutawa et al. [2], and implemented measures for additional properties of intra-version ASs and versions. For detecting supercycles, we employed Tarjan's [33] algorithm for identifying strongly connected components. Arcan natively detects subcycles based on depth-first search according to Sedgewick and Wayne [30]. Consequently, every subcycle can be matched to its supercycle in constant time complexity by selecting any of the subcycle's components and checking which supercycle affects the component, as there is a 1:n relationship.

We eventually realized that Arcan requires by far the most processing time in our tool chain. This is why we restructured Arcan to support multi-threading by changing static variables and singleton classes with a state into instance variables and regular classes. Furthermore, we optimized Arcan by reducing the number of repeated dependency graph traversals through increased storing of intermediate results.

B. AsTdEA

We implemented AsTdEA as an open-source Java program that is available both as source code and a ready-to-run compiled download.[1] AsTdEA is structured into a data, logic, and I/O layer. It executes Arcan, which is provided in a separate .jar file, by calling Arcan's main method. Based on the number of processors available to Java's virtual machine, AsTdEA creates several threads that each run Arcan for a particular version that shall be analyzed. Each thread stores its results in own .csv files. Only after analyzing all versions of a system, we load all generated .csv files again to perform the remaining computations for our inter-version ASs tracking and properties. This separation enables modular changes to the toolchain and the independent execution of Arcan and AsTdEA. The tracking of ASs is based on determining the identity of affected components in adjacent versions. For this purpose, similar to the related tool ASTracker (cf. Section VII), we compare their fully qualified names (i.e., package.class).

We ensured the implementation's quality by creating unit tests for central functionalities, such as the tracking of ASs. Furthermore, we simulated the evolution of entire systems and

[2] https://gitlab.com/essere.lab.public/arcan

[3] https://github.com/PhilippGnoyke/arcan-1.2.1-modded

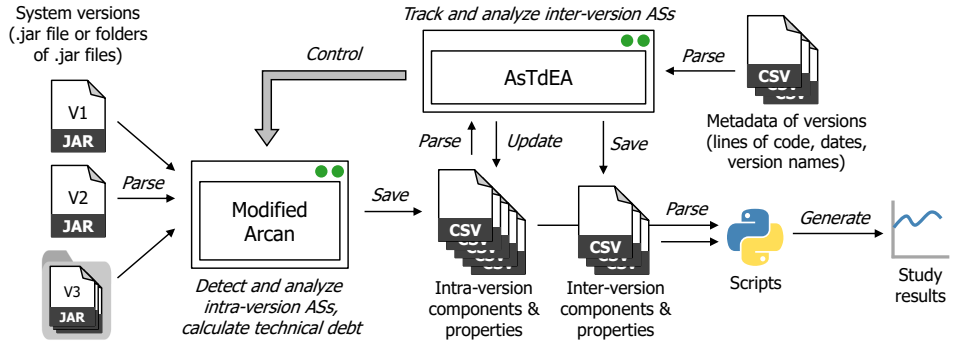

Fig. 2. Data flow in our toolchain.

conducted studies on real-world systems [12] to verify that `AsTdEA` correctly identifies, tracks, and quantifies the properties of all ASs. We encourage researchers and practitioners to raise issues, create pull requests, and fork our repositories to increase `AsTdEA`'s value for future users.

C. Scripts

For our previous studies, we queried the generated data from `AsTdEA` by creating Python scripts that aggregate the data, perform statistical tests, and generate visualizations. As a further contribution, we provide all of these scripts within our repository.[1] Moreover, for each of our studies, we published the respective versions of `AsTdEA` we used in a persistent open-access repository on Zenodo.

V. Challenges

While developing `AsTdEA` and modifying `Arcan`, we faced and resolved several challenges. Next, we share our experiences and lessons learned from building on existing research tools to benefit future academic tool development—especially in the area of ASs as well as code smells. Afterwards, we summarize issues that were not trivial to solve, and are still persisting within `AsTdEA`. Currently, we are working on resolving such issues, for which we sketch a broader outlook in Section VI.

A. Lessons Learned

Comprehending code that has been written by other developers can be challenging [16], [31]. Accordingly, comprehending `Arcan`'s code to the level of confidently modifying it required considerable effort—especially given a different coding style, partially long classes, and the heavy use of external libraries. Building on our personal experiences, we recommend other researchers to plan in enough time when undertaking similar efforts and testing any changes they implement extensively. Depending on the personal experience, it can be very valuable to systematically familiarize oneself with different build tools and pipelines such as `Maven`, especially when external libraries are involved (`Arcan` alone has 217).

Initially, we conceptualized `AsTdEA` as a static tool for analyzing past versions of systems in a batch mode. Later, we intended to integrate `AsTdEA` into pipelines for continuously checking code quality, which required running `Arcan` incrementally on new versions. However, analyzing the entire system history in batch mode for every new version has turned out a considerable waste of time and resources, which is why we adapted `AsTdEA` to allow the parsing of `Arcan` output without running it again. This required the restructuring of fundamental entry points into the program and represents a classical example of software having to adapt to changing requirements [17], [36]. Such situations showcase the importance of modular and non-monolithic design from the get-go, as this allows components to be easily modified and exchanged without impacting wide parts of the system, even for initially small academic tools. So, we recommend to structure future academic tools into concise units of functionality that can be modified in more isolation.

B. Persisting Challenges

The primary challenge persisting within our toolchain is the efficient use of `Arcan`, whose runtime (as indicated in Section IV-A) is usually several orders of magnitude slower than `AsTdEA`'s. We observed a highly non-linear scaling of the execution time with the system size. In our experiments, this ranged from seconds for smaller systems to hours for larger ones. This is not surprising given that creating dependency structures and traversing them for various AS detection and metric computation purposes becomes increasingly complex for larger systems. Partially due to this phenomenon, we previously excluded the system `Eclipse` from our empirical analyses of the Qualitas Corpus Evolution Distribution dataset [35]. Moreover, the runtime problem impedes running a fine-grained evolutionary analysis on many snapshots of a system to gain more accurate insights. We present our ideas for mitigating this issue in Section VI-A.

Another challenge concerns the tracking accuracy of `AsTdEA` when a considerable share of components is renamed (cf. Section III-A)—even if the renaming affects only the package structure. Specifically, if enough components of an intra-version AS are renamed in the subsequent version, including the central component of HDs and UDs, it cannot be linked to related intra-version ASs. This represents false-negative tracking results and incorrectly increases the removal and introduction rates of inter-version ASs. Furthermore, statistical analyses like time series of properties or survival

analyses are confronted with noisy data. So far, we have addressed this problem by performing manual analyses and validations of AsTdEA's output. In Section VI-C, we discuss possible solutions for improving the tracking accuracy for ASs.

The last point relates to changing requirements as outlined in Section V-A. Over time, our scripts to extract data from the output of AsTdEA have grown considerably in size and complexity, which makes them increasingly error-prone, hard to maintain, and hard to extend further for new analyses. Building on related insights into scripting [29], we describe our strategy for mitigating this trend in the future within Section VI-D.

VI. Outlook

To address the persisting challenges we described in Section V-B, we now sketch a series of ideas we plan to realize to overcome these challenges. While these future directions are based on our experiences and limited to AsTdEA as part of our own future work, other researchers working on ASs may benefit form these ideas to develop or improve their own techniques—or feel inspired to propose solutions for these.

A. Accelerating Arcan

A central goal of improving our toolchain is to speed up Arcan. While we already performed some optimizations like multi-threading (cf. Section IV-A), we argue that there is a lot of potential for improvement left. For this purpose, we first want to critically analyze what procedures in Arcan are consuming the most time and how they could be optimized. Possible bottlenecks are the parsing of .jar files, inefficiencies in TinkerPop queries (which Arcan is built upon), or redundant graph traversals. In case we identify improvable bottlenecks, we will update our modification of Arcan in our future work. Meanwhile, we realize that certain algorithms in property calculation and smell detection tend to scale non-linearly and limit the potential for optimization. Therefore, a possible solution could be to avoid re-performing every graph traversal with every analyzed version. Especially when we frequently analyze snapshots instead of releases, adjacent versions will only differ marginally from one another. Thus, we want to develop strategies to identify the set of changed components and limit graph traversals to as few components as possible without sacrificing accuracy. For instance, if no dependencies have been added to or removed from a CD's affected components, we can exclude its section of the dependency graph from both algorithms for detecting supercycles and subcycles.

B. Enhancing the Output of Arcan

While working on Arcan, we also want to enhance its output. Currently, from a user point of view, the information on dependencies between classes does not discriminate whether it represents static method calls, member variables, parameters in methods, or anything else. Furthermore, Arcan does not output the strength of dependencies, specifically in how many contexts the dependent class references the depending class. Within Arcan, information on whether an edge represents an inheritance relationship exists, which we aggregate as a property of class-level CDs, but this is not on a fine-grained level. Instead, we aim to provide more extensive information on the type and strength of dependencies—both within the source code for further computations and as .csv output. The former can be useful to quantify how easy the refactoring of intra-version ASs can be, as it can be assumed that loosely coupled dependency edges are easier to remove. We presume that this information could be especially useful for analyzing CDs, because, in some cases, breaking a single edge can resolve an entire supercycle or at least cause a split into more manageable, less severe CDs.

C. Increasing Tracking Accuracy

The next challenge, which we discussed in Section V-B, relates to the tracking accuracy of ASs that comprise renamed components. We aim to reduce the share of false-negative mappings by changing how we determine the identity of components. While we can maintain comparing full names as a primary step (and as is done in related work), we intend to include a secondary step that is based on code clone detection [26]. While code clone detection primarily serves purposes like identifying technical debt in the form of copied code segments—which are considered to reduce maintainability (especially when faulty code has been copied), detecting plagiarism, or comprehending a system's source code evolution [14], we can also employ it for our use case. Essentially, we have to compare smell-affected classes whose intra-version ASs of the same type have not been matched yet with another intra-version AS. For HDs and UDs, this can again happen in a two-stage process, where the central components are compared first to not harm AsTdEA's performance. For CDs, it may be sufficient to take a sample of a few classes given our matching criterion of two shared components. We will test different techniques and measure their tracking accuracy, precision, and recall against manually matched classes.

D. Reducing the Complexity of Scripts

After having performed multiple empirical studies using AsTdEA's output, we developed a more precise conception about what parts of that data are useful. We intend to reduce the complexity of subsequent scripts by including the computation of more metrics and properties, as well as providing the same data in alternative ways that reduce the need for aggregating. By doing so, we can limit scripts to statistical tests and visualizations, which reduces inter-dependencies of script functions and ensures the scripts' maintainability. Eventually, we want to make AsTdEA more customizable by dividing computations and output into modules that can be switched on and off based on the user's requirements—all while maintaining an easy access to AsTdEA with default parameter values.

VII. Related Work

The most similar tool to AsTdEA is ASTracker[4] (Architecture Smell Tracker) developed by Sas et al. [27]. It also executes Arcan—but a version that generates .graphML

[4]https://github.com/darius-sas/astracker

files, which contain the dependency graph of each analyzed version, with ASs being represented as vertices pointing to affected components. `ASTracker` tracks smells by performing a Jaccard set comparison of the affected components of each AS of the same type. This is related to our `AsTdEA`, as we outlined in Section III-A. However, we expanded this concept by including a primary step of comparing the central components of HDs and UDs. Moreover, Sas et al. detect CDs as subcycles and also link them via set comparison, while our supercycle-based approach only requires two matching components to link two intra-version CDs. In both tools, the identity of components is determined by comparing their full names—which we plan to extend though (cf. Section VI-C).

VIII. Conclusion

In this paper, we presented our toolchain for tracking ASs and computing various of their properties, consisting of a modified version of `Arcan` for detecting intra-version ASs and our own tool `AsTdEA`, which executes `Arcan` and connects identified ASs as inter-version ASs. We highlighted the use case of `AsTdEA` by summarizing its usage, its output, and the novel concepts on the evolutionary analysis of ASs that we implemented. By providing lessons learned based on challenges we faced during the development of `AsTdEA`, such as modifying existing code or changing requirements, we hope to ease the creation of other academic tools. Furthermore, we shared open challenges and how we intend to solve them in order to inspire research in the domain of ASs. This includes, for instance, accelerating and enhancing `Arcan`, improving the tracking accuracy by introducing code-clone detection techniques, or reducing script complexity by extending `AsTdEA`'s output.

References

[1] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An Empirical Study of Design Degradation: How Software Projects Get Worse Over Time," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015.

[2] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, "On the Shape of Circular Dependencies in Java Programs," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE, 2014.

[3] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A Tool for Architectural Smells Detection," in *Proceedings of the International Conference on Software Architecture (ICSA)*. IEEE, 2017.

[4] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic Detection of Instability Architectural Smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016.

[5] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. B. Seaman, "Managing Technical Debt in Software Engineering," *Dagstuhl Reports*, vol. 6, no. 4, 2016.

[6] U. Azadi, F. Arcelli Fontana, and D. Taibi, "Architectural Smells Detected by Tools: A Catalogue Proposal," in *Proceedings of the International Conference on Technical Debt (TechDebt)*. IEEE, 2019.

[7] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[8] T. Besker, A. Martini, and J. Bosch, "Technical Debt Cripples Software Developer Productivity," in *Proceedings of the International Conference on Technical Debt (TechDebt)*. ACM, 2018.

[9] W. Cunningham, "The WyCash Portfolio Management System," *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, vol. 4, no. 2, 1992.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.

[11] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying Architectural Bad Smells," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009.

[12] P. Gnoyke, S. Schulze, and J. Krüger, "An Evolutionary Analysis of Software-Architecture Smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.

[13] I. Gorton, *Essential Software Architecture*. Springer, 2011.

[14] K. Inoue and C. K. Roy, *Code Clone Analysis: Research, Tools, and Practices*. Springer, 2021.

[15] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, 2012.

[16] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do You Remember This Source Code?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2018.

[17] E. Kuiter, J. Krüger, and G. Saake, "Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia," in *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2021.

[18] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[19] R. C. Martin, "OO Design Quality Metrics: An Analysis of Dependencies," 1994.

[20] A. Martini, F. Arcelli Fontana, A. Biaggi, and R. Roveda, "Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company," in *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer, 2018.

[21] H. Melton and E. D. Tempero, "An Empirical Study of Cycles among Classes in Java," *Empirical Software Engineering*, vol. 12, no. 4, 2007.

[22] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," in *Proceedings of the Working Conference on Software Architecture (WICSA)*. IEEE, 2015.

[23] D. L. Parnas, "Software Aging," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1994.

[24] R. Roveda, "Identifying and Evaluating Software Architecture Erosion," Ph.D. dissertation, University of Milano-Bicocca, 2018.

[25] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an Architectural Debt Index," in *Proceedings of the International Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018.

[26] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.

[27] D. Sas, P. Avgeriou, and F. Arcelli Fontana, "Investigating Instability Architectural Smells Evolution: An Exploratory Case Study," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.

[28] D. Sas, P. Avgeriou, and U. Uyumaz, "On the Evolution and Impact of Architectural Smells: An Industrial Case Study," *Empirical Software Engineering*, vol. 27, no. 4, 2022.

[29] S. Schulze and W. Fenske, "Analyzing the Evolution of Preprocessor-Based Variability: A Tale of a Thousand and One Scripts," in *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2018.

[30] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley, 2011.

[31] D. Spinellis, *Code Reading: The Open Source Perspective*. Addison-Wesley, 2003.

[32] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier, 2014.

[33] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.

[34] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2010.

[35] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2010.

[36] K. Wiegers and J. Beatty, *Software Requirements*. Microsoft Press, 2013.

[37] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the Impact of Design Debt on Software Quality," in *Proceedings of the International Workshop on Managing Technical Debt (MTD)*. ACM, 2011.