ARCADE: An Extensible Workbench for Architecture Recovery, Change, and Decay Evaluation

Marcelo Schmitt Laser
Nenad Medvidovic
(schmittl,neno)@usc.edu
University of Southern California, USA

Duc Minh Le
dle50@bloomberg.net
Bloomberg LP, USA

Joshua Garcia
joshug4@uci.edu
University of California, Irvine, USA

ABSTRACT

This paper presents the design, implementation, and usage details of ARCADE, an extensible workbench for supporting the recovery of software systems' architectures, and for evaluating architectural change and decay. ARCADE has been developed and maintained over the past decade, and has been deployed in a number of research labs as well as within three large companies. ARCADE's implementation is available at https://bitbucket.org/joshuaga/arcade and the video depicting its use at https://tinyurl.com/arcade-tool-demo.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; *Software system models*; *Software maintenance tools*; **Software evolution**.

KEYWORDS

software architecture, tool, maintainability, reverse engineering, software decay

ACM Reference Format:
Marcelo Schmitt Laser, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia. 2020. ARCADE: An Extensible Workbench for Architecture Recovery, Change, and Decay Evaluation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417941

1 INTRODUCTION

Architecture plays a critical role in managing the evolution of large, complex, and long-lived software systems [1, 22, 25, 26]. One of the most impactful evolutionary aspects of a software system is *architectural drift and erosion*, i.e., the inclusion of design decisions other than those originally intended, which are referred to collectively as *architectural decay*. Such decay manifests as architectural smells [8, 9], software anti-patterns [3], and technical debt [14]. An in-depth understanding of a system's architecture is therefore critical to effective evolution management.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417941

Attaining such an understanding is difficult: the architectures of large systems are often obscured and human-made documentation is unreliable [27]. This problem is compounded when managing multiple versions of a software system [7]. To enable continuous analysis of a software system's evolution, a framework is needed for the automatic recovery and evaluation of its architecture.

Analysis of a software system's architecture can be divided into at least five principal tasks: (1) recovery, (2) decay detection, (3) measurement, (4) visualization, and (5) prediction. These tasks have guided our work and form the underpinning for the tool suite we present in this paper. Architecture *recovery* is the process of building architectural models from a system's implementation artifacts [7]. *Decay detection* analyzes these models for the presence of architectural "smells", manifestations of poor design decisions in a system's architecture that negatively impact its lifecycle properties, such as understandability, extensibility, and reusability [8, 9]. An important aspect of architectural *measurement* is the quantification of architectural decay based on characteristics of the recovered models [2]. Architectural *visualization* is the presentation of important information from the architectural model in a human-understandable form [21]. Finally, *prediction* is the use of models to detect the architectural significance of future system issues and modifications [23].

While a number of software tools are available for each of the various tasks related to software architecture analysis [6, 12, 21], there have been no solutions available that perform all of the tasks in tandem. Furthermore, existing tools are often difficult to use out-of-the-box, restricting their adoption. Once adopted, they can be difficult to combine due to using different data and file formats. This motivated us to present ARCADE – Architecture Recovery, Change, and Decay Evaluator. ARCADE is an extensible research workbench that incorporates a large number of both custom-built and third-party tools. ARCADE allows the inclusion of new solutions for the tasks involved in software architecture analysis. Its execution is flexible, allowing the user to select which tasks to execute and which of the tool options to apply for each task, including the possibility of executing multiple tools for any single task. For example, a user may choose to execute only recovery and decay detection on their system, but to recover multiple different architectural models by applying different recovery tools. Importantly, ARCADE inherits the properties of the tools it implements and/or incorporates off-the-shelf, allowing one to select different trade-offs in their analysis depending on the tools executed. It also allows direct empirical comparison of the selected tools.

ARCADE is a relatively large research-off-the-shelf system developed and evolved over the past decade. It has been deployed at over 20 research groups around the world and at three large companies.

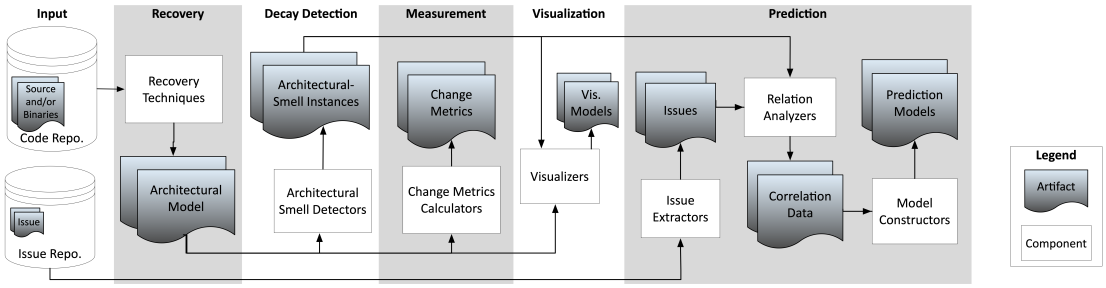

Figure 1: ARCADE's dataflow architecture, with its inputs and five principal subsystems delineated in different vertical segments. Each subsystem has at least one component and produces at least one artifact.

In our research group, ARCADE has been applied in the analysis of nearly 1,000 versions of over 50 software systems, totaling around 500 MSLOC. This paper describes ARCADE's design, its support for the five principal architectural analysis tasks described above, and the details of its implementation and application in practice.

2 DESIGN

ARCADE's high-level architecture is divided into five subsystems, focusing on the principal tasks of (1) recovery, (2) decay detection, (3) measurement, (4) visualization, and (5) prediction. Each of these tasks may be performed by one or more tools integrated within ARCADE. The architecture of ARCADE is shown in Figure 1.

2.1 Recovery

Recovery constructs architectural models from implementation-level artifacts. ARCADE allows the integration of any architecture recovery technique, so long as it is encapsulated to be invokable using a standardized API. ARCADE currently integrates ten off-the-shelf recovery tools [7], including ACDC (Algorithm for Comprehension-Driven Clustering) [27], ARC (Architecture Recovery using Concerns) [10], WCA (Weighted Combined Algorithm) [18], Bunch [17], and ZBR (Zone-Based Recovery) [4, 5]. Each of these tools may be run independently or in tandem, enabling the recovery of a system's architecture from a specific perspective or the recovery of multiple perspectives at once; they may also be run in batches, allowing simultaneous architecture recovery of multiple versions of a system, or even multiple systems.

The recovery tools integrated within ARCADE implement different strategies for clustering implementation-level entities into architectural elements, including dependency analysis, information retrieval, search-based strategies, machine learning, etc. The details of the clustering depend on the tool and vary at multiple levels, such as granularity (e.g., the kinds of code-level entities that are clustered) and viewpoint (e.g., clustering based on patterns vs. concerns). For example, ACDC identifies certain patterns in the dependency structures of a system's implementation classes, grouping them in a way that minimizes inter-component dependencies [27]. The recovery techniques also represent different properties in the resulting architectures (e.g., semantic as opposed to syntactic properties) [7], making the combination of their results potentially advantageous.

All recovery tools in ARCADE are applicable to systems built in C/C++ and Java; a subset of the tools is also available for C# and Python. Our particular focus on supporting C/C++ and Java was driven both by their widespread use in practice and by the availability of off-the-shelf static analyzers for them. The tools take as input either source code or binaries. The output of a recovery tool is a list of dependencies and a list of clusters, which together represent the components and connections of a software system.

2.2 Decay Detection

Architectural smells are instances of potentially problematic design decisions that, over time, cause architectural decay [8, 9]. ARCADE's decay detectors are applied on the outputs of the recovery tools to identify those instances. Decay detectors currently available through ARCADE are capable of identifying 11 different architectural smells, grouped in four categories: interface-based, change-based, concern-based, and dependency-based [14].

Interface-based smells are defined using the relationship between the number of interfaces exported by a given component and the number of other system components that depend on these interfaces. Change-based smells represent instances of components that often must be modified at the same time during the system's life span. Concern-based smells are occurrences of undesirable semantic issues, such as multiple components performing the same function; these smells are detected within ARCADE using natural language processing techniques, such as topic modeling. Finally, dependency-based smells represent problems in the dependency graph of a software system, such as cycles or an excessive number of dependencies involving a particular component. The list of architectural smells currently detected by ARCADE is found in Table 1.

While all decay detection tools utilize the results of architecture recovery as input, some require specific recovery tools to be used, and some may require additional information beyond that provided by the recovery tools. For example, concern-based smell detection tools require topic models that are only generated by some of ARCADE's recovery techniques (e.g., ARC [10]). Meanwhile, change-based smell detection tools require historical data (e.g., logs) from a version-control repository, as well as models of multiple versions of a system. The output of a decay detection tool is a list of smell instances and their affected components and/or code-level entities.

2.3 Measurement

ARCADE incorporates various metrics for quantifying architectural change and decay. These metrics focus either on the whole system or on individual components (i.e., the recovered clusters of code-level entities). Some of these metrics are described below.

Table 1: Architectural smells detected by ARCADE

Category	Type	Definition	Consequences
Interface-based	Unused Interface	Component's interface is not linked to other components	Adds unnecessary complexity to the system
	Unused Brick	Component's interfaces are all unused	Same as Unused Interface, but more severe
	Sloppy Delegation	Component delegates functionality it could have performed	Reduces separation of concerns
	Functionality Overload	Component has an excessive amount of functionality	Reduced modularity
	Lego Syndrome	Component handles exceedingly small amount of functionality	High coupling
Change-based	Duplicate Functionality	Several components replicate the same functionality	Bugs if changing only one duplicate
	Logical Coupling	Parts of different components are frequently changed together	Similar to Duplicate Functionality
Dependency-based	Dependency Cycle	Set of components whose links form a circular chain	Changes to one component affect the entire cycle
	Link Overload	Component's interfaces have too many dependencies	Reduced isolation of changes
Concern-based	Scattered Parasitic Funct.	Multiple components responsible for realizing one concern	Changing a feature modifies multiple system parts
	Concern Overload	Component implements an excessive number of concerns	Violates separation of concerns

2.3.1 System-Level Metrics. System-level metrics in ARCADE are used to quantify characteristics of an entire architecture. They support the measurement of change and decay aspects of multiple versions of a system, helping to assess its evolutionary traits [2, 14].

MojoFM [28] is a distance metric between two architectures, and is based on two operations used to transform one architecture into another: moves (*Move*) of implementation-level entities from one architectural cluster to another and merges (*Join*) of clusters.

a2a measures architectural change [2] by computing the *minimum* number of operations required to transform one architecture to another. This transformation is based on five fundamental operations on an architecture: adding/removing a cluster (i.e., component), adding/removing a code-level entity to/from a cluster, and moving an entity between clusters. *a2a* is computed using the Hungarian algorithm [20], to account for the fact that the total number of code-level entities in the architecture changes over time.

Finally, *cvg* [2] measures the similarity of clusters within an architecture. The similarity between each pair of clusters is calculated using a component-level metric, *c2c*, which represents the proportion of code-level entities that overlap between the clusters (discussed further below).

2.3.2 Component-Level Metrics. Component-level metrics measure aspects of specific components within an architecture, as well as the relationship between groups of components.

c2c [2] measures the degree of overlap between implementation-level entities contained within two clusters:

$$c2c(c_m, c_n) = \frac{|c_m \cap c_n|}{max(|c_m|, |c_n|)} \times 100\%$$

The denominator normalizes the entity overlap, ensuring that *c2c* provides the most conservative value of cluster similarity.

Modularization Quality (*MQ*) [19] spans the component- and system-levels. It is a measure of coupling and cohesion, defined as:

$$MQ = \sum_{i=1}^{k} CF_i$$

k is a system's number of clusters. CF_i is the "cluster factor" of cluster i, representing i's coupling and cohesion defined as

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j \neq i}}^{k} (\epsilon_{i,j} + \epsilon_{j,i})} & otherwise \end{cases}$$

μ_i is the number of edges within the cluster, measuring cohesion; $\epsilon_{i,j}$ is the number of edges from cluster i to j, measuring coupling.

2.4 Visualization

The information obtained through architectural analysis helps engineers answer questions related to the current state and evolution of an architecture, i.e., how it changes and decays. These results can be difficult to comprehend, however, due to the overwhelming amount and nature of information (structured text and numbers). Without adequate visualization support, engineers wanting to investigate a specific change or smell would have to search manually through results spanning multiple recovered and analyzed architectural models. This impacts the understandability of ARCADE's output and can actually hamper the needed maintenance activities.

To facilitate the comprehension of ARCADE's results, a visualization tool is therefore needed. Such tools select and display information from models in a human-understandable, often graphical, format, in accordance with a particular viewpoint.

Two visualization tools are currently available in ARCADE: EVA [21] and ArcadeViz [13]. EVA, shown in Figure 2, graphically represents the clusters obtained by a recovery tool as large circles containing smaller circles for code-level entities. EVA can display the cluster sets of multiple versions of a system simultaneously. This allows the user to visualize and explore the system's architecture, as well as gauge the impact of particular design decisions and judge the system's architectural stability.

ArcadeViz, shown in Figure 3, uses active views and color-coded labels to develop views based on the D3.js library. ArcadeViz focuses on visualizing the architectural differences between versions, including structural changes in an architecture, changes of components' dependencies and interfaces, and specific locations of decay.

2.5 Prediction

While architectural decay may not manifest itself in outright system failures, it imposes real costs in terms of engineers' time and effort (e.g. technical debt), and can negatively impact system reliability and performance. Over time, those implicit and disguised problems may be revealed as explicit implementation issues.

ARCADE leverages the correlations between symptoms of architectural decay and reported implementation issues to accurately predict a system's characteristics that will likely manifest in a future version (e.g., new issues or increased proneness to change). ARCADE's Issue Extractor and Relation Analyzer are used to collect

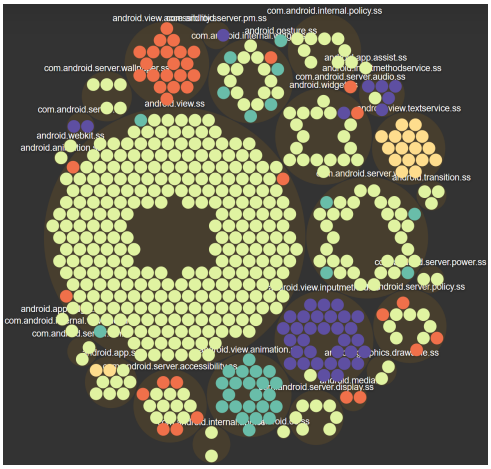

Figure 2: EVA visualization of the recovered architecture of Android 8.0's Application Framework subsystem. Colors indicate code-level entities from different packages.

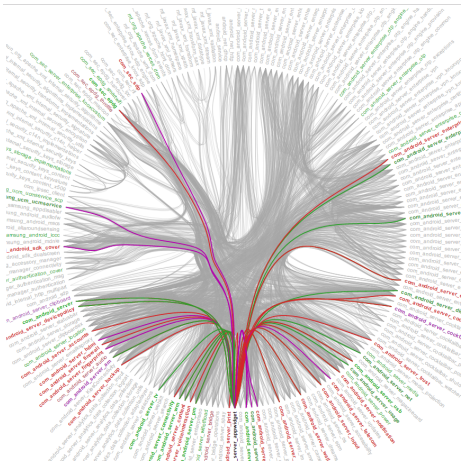

Figure 3: Example ArcadeViz visualization of dependency changes between versions 7.0 and 8.0 of Android. Red lines indicate incoming dependencies, green lines outgoing dependencies, and purple lines removed dependencies.

the underlying raw system-issue and architectural-smell data, preprocess it, and feed it to ARCADE's Model Constructor (Figure 1).

To date, we have implemented two instances of the Model Constructor. One instance leverages WEKA [11], a well-known ML framework, to build issue- and change-proneness prediction models for a system and to evaluate the models' accuracy [15]. The other builds models to estimate the architectural significance of newly reported implementation issues based on historical data [24]. Both of these are examples of capabilities enabled by ARCADE that can help engineers to prevent the adverse effects of architectural decay.

3 ARCADE IN PRACTICE

ARCADE has been developed and evolved over the past decade. Its current implementation totals over 100 KSLOC distributed across several languages. In this section, we tie together its major elements described above and summarize its use to date.

3.1 Typical Usage Scenario

A typical application of ARCADE begins with downloading the source code and/or binaries of a subject system, and extracting the implementation issues from an issue repository. One or more recovery tools are then applied to the implementation artifacts, producing models of the subject system's architecture for each of the system's versions. These models consist of collections of architectural clusters, which represent the system's components, and of all dependencies between the system's implementation entities.

Next, decay detection tools are executed over the recovered architectural models, generating lists of architectural smells. Simultaneously, the architectural models are used by measurement tools to quantify different aspects of architectural quality, producing the values of various metrics relating to architectural change and decay. All of these results may be visualized in multiple different ways, each focusing on a particular set of viewpoints.

Finally, the prediction tools correlate the subject system's issues with its detected smell instances, creating a dataset that is used to build predictive models of the system's architectural quality and evolution. These models offer a range of information, from the architectural significance of particular issues to the likelihood of certain smells appearing in certain components.

3.2 Application and Use To Date

ARCADE has been deployed at over 20 universities worldwide and at three large companies. It has been used to analyze software systems of up to 10 MSLOC (namely, Chromium) [16]. We estimate the amount of code it has analyzed to date within the studies conducted by our team alone to be on the order of 500 MSLOC. ARCADE's visualization tools have been used to depict the architectures of systems as large as Android OS's Application Framework [21].

ARCADE's results have been validated by the architects of multiple Apache Foundation projects, as well as engineers within the aforementioned companies. While we are aware of these external deployments of ARCADE, we either do not have detailed information on its use or are prevented from reporting it due to the proprietary IP involved. ARCADE has also been used to analyze itself, and that analysis has helped greatly with maintaining and evolving its own code base. Finally, ARCADE has been used in multiple instances in classroom assignments, aiding in teaching software architecture concepts to graduate-level students.

4 CONCLUSION

ARCADE is a large research-off-the-shelf system whose scope has grown gradually from architecture recovery alone to the range of capabilities described in this paper. In addition to its widespread use, ARCADE has played a key role in at least four completed and one ongoing doctoral dissertation. The resulting changes applied to it by different developers for their specific needs have led to ARCADE's own design decay, and a significant portion of it has recently undergone a major refactoring, aided by ARCADE itself.

ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation under grants 1717963, 1823354, and 1823262 and U.S. Office of Naval Research under grant N00014-17-1-2896.

REFERENCES

[1] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software architecture in practice*. Addison-Wesley Professional.

[2] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2017. A Large-Scale Study of Architectural Evolution in Open-Source Software Systems. *Empirical Softw. Engg.* 22, 3 (June 2017), 1146–1193. https://doi.org/10.1007/s10664-016-9466-0

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

[4] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. 2011. Investigating the use of lexical information for software system clustering. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 35–44.

[5] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. 2010. A probabilistic based approach towards software system clustering. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 88–96.

[6] Eduardo Figueiredo, Jon Whittle, and Alessandro Garcia. 2009. ConcernMorph: Metrics-Based Detection of Crosscutting Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 299–300. https://doi.org/10.1145/1595696.1595751

[7] J. Garcia, I. Ivkovic, and N. Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 486–496.

[8] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 255–258.

[9] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a catalogue of architectural bad smells. In *International Conference on the Quality of Software Architectures*. Springer, 146–162.

[10] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 552–555.

[11] G. Holmes, A. Donkin, and I.H. Witten. 1994. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 357–361.

[12] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. 2019. ENRE: A Tool Framework for Extensible ENtity Relation Extraction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 67–70. https://doi.org/10.1109/ICSE-Companion.2019.00040

[13] Duc Minh Le. 2018. *Architectural Evolution and Decay in Software Systems*. Ph.D. Dissertation. University of Southern California.

[14] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. 2016. Relating Architectural Decay and Sustainability of Software Systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 178–181.

[15] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. 2018. An Empirical Study of Architectural Decay in Open-Source Software. In *2018 IEEE International Conference on Software Architecture (ICSA)*. 176–17609.

[16] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger. 2018. Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques. *IEEE Transactions on Software Engineering* 44, 2 (2018), 159–181.

[17] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. 1999. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 50–59.

[18] Onaiza Maqbool and Haroon Babri. 2007. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering* 33, 11 (2007), 759–780.

[19] B. S. Mitchell and S. Mancoridis. 2006. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering* 32, 3 (2006), 193–208.

[20] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.

[21] D. Nam, Y. K. Lee, and N. Medvidovic. 2018. EVA: A Tool for Visualizing Software Architectural Evolution. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 53–56.

[22] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes* 17, 4 (1992), 40–52.

[23] A. Shahbazian, D. Nam, and N. Medvidovic. 2018. Toward Predicting Architectural Significance of Implementation Issues. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 215–219.

[24] Arman Shahbazian, Daye Nam, and Nenad Medvidovic. 2018. Toward Predicting Architectural Significance of Implementation Issues. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 215–219. https://doi.org/10.1145/3196398.3196440

[25] Mary Shaw, David Garlan, et al. 1996. *Software architecture*. Vol. 101. prentice Hall Englewood Cliffs.

[26] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.

[27] Vassilios Tzerpos and Richard C Holt. 2000. Accd: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 258–267.

[28] Zhihua Wen and Vassilios Tzerpos. 2004. An effectiveness measure for software clustering algorithms. In *International Workshop on Program Comprehension (IWPC)*. IEEE.