Capturing and Understanding the Drift Between Design, Implementation, and Documentation

Joseph Romeo
REVEAL @ Software Institute – USI, Lugano, Switzerland
joseph.romeo@alumni.usi.ch

Marco Raglianti
REVEAL @ Software Institute – USI, Lugano, Switzerland
marco.raglianti@usi.ch

Csaba Nagy
REVEAL @ Software Institute – USI, Lugano, Switzerland
csaba.nagy@usi.ch

Michele Lanza
REVEAL @ Software Institute – USI, Lugano, Switzerland
michele.lanza@usi.ch

ABSTRACT

UML artifacts constitute a key (but often neglected) asset supporting the comprehension of a system. Design documents "bind" developers in implementation phases and close the loop as documentation of the implemented system itself. Nevertheless, the intended system (design), its current version (implementation), and its documentation, naturally tend to drift apart, negatively impacting the usefulness of UML diagrams contained in such artifacts.

We present a novel approach to capture and understand the *Design–Implementation–Documentation (DID) drift*. We connect UML references in human-readable text-based UML formats (*e.g.,* PlantUML) to the corresponding source code entities (*e.g.,* Java classes), implementing novel metrics to capture the UML coverage of the system. We analyze project and file coverage evolution across releases and commits, with overall, method-level, and attribute-level detailedness, showing how they support DID drift analysis. We present interesting case studies exemplifying how through Drifter, the visual exploration tool we developed to validate our approach, we identify DID drift and ways to tackle it in the future.

CCS CONCEPTS

• **Software and its engineering** → **Designing software**; **Documentation**; Abstraction, modeling and modularity.

KEYWORDS

design implementation documentation drift, DID drift, UML

ACM Reference Format:
Joseph Romeo, Marco Raglianti, Csaba Nagy, and Michele Lanza. 2024. Capturing and Understanding the Drift Between Design, Implementation, and Documentation. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643916.3644399

Acknowledgments. This work is supported by the Swiss National Science Foundation project "INSTINCT" (Project No. 190113).

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04…$15.00
https://doi.org/10.1145/3643916.3644399

1 INTRODUCTION

The Unified Modeling Language (UML) [13] is a visual modeling language that provides a standard way to describe (not only) software systems through a number of specialized diagrams. UML is used for design and documentation, providing a guideline for system development and evolution, thus constituting a fundamental support for the comprehension of the system's architecture and behavior, especially at higher levels of abstraction.

As software systems evolve and grow in size and complexity, they naturally diverge from their intended architecture. In the literature we can find multiple definitions circumscribing this phenomenon: Architecture erosion [4, 9, 19–21], architecture degradation [15], architecture consistency [2, 23], or architecture recovery [10] when attempting to mitigate the effect of an inevitable drift between the intended architecture and the implementation of the system.

These definitions miss an important aspect of the software development lifecycle. When a system is implemented, its current state must be documented and, although documentation is often an afterthought [1], design artifacts are the starting point for a high level comprehension. In practice, what distinguishes a design UML diagram from one to document the system, is time: Design-phase artifacts precede implementation, for example to describe a new feature, while a UML artifact conceptually becomes documentation only after it reflects the implemented architecture.

We investigate the relationships between design, implementation, and documentation. We model entities in two domains, UML diagrams and source code, to analyze how their artifacts drift apart, creating a gap between the UML representation and the actual implementation (*e.g.,* classes covered by a diagram). We leverage the ease of parsing and the popularity of PlantUML, a text-based human-readable UML format, to associate Java entities (*e.g.,* classes, interfaces) with the UML diagrams mentioning them.

Structural relationships among entities and temporal relationships among activities on artifacts of the two domains (*e.g.,* file commits in the repository), are a form of drift "in space and time", providing insights on the nature, origin, and extent of the phenomenon we call *Design–Implementation–Documentation (DID) drift.*

2 DID DRIFT

We define Design–Implementation–Documentation (DID) drift as the distance between design and implementation and that between implementation and documentation. There are two types of DID drift: Space DID drift, stemming from the coverage metrics between UML and source code entities, and time DID drift, when we consider

the temporal difference between modifications to artifacts that alter such coverage metrics, trying to capture how drift evolves.

We distinguish between overall coverage, and two detailedness metrics capturing finer-grained information: Attribute-level detailedness, and method-level detailedness.

Overall Coverage. The percentage of Java entities (classes, interfaces, enums) present in at least one UML diagram.

Attribute-level Detailedness. The percentage of attributes in a Java class present in at least one UML diagram.

Method-level Detailedness. The percentage of methods in a Java class present in at least one UML diagram.

For detailedness, if a method or attribute is covered in multiple diagrams, the percentage can be the minimum, maximum, or average of the respective detailedness for each UML diagram.

3 DATASET

Using the SEART GitHub Search (GHS) tool [7] we selected projects, excluding forks, with at least 2,000 commits, 10 contributors, 100 stars, and 10k *cloc* [8] LOCs to remove toy projects. We had a starting dataset of 13,152 repositories, which we cloned locally.

UML File Tagging. File extensions are an efficient way to discriminate file types, but not sufficient to uniquely identify UML diagrams. Thus, to identify UML artifacts, we combined: Import/export extensions from tools returned by searching for "*top UML diagramming tools*" on Google, all extensions with "`uml`" in the name and all extensions in any file path with "`uml`" in the name, when present in at least two repositories. We removed any known non-UML extension (*e.g.,* .java, .jar, .am). We implemented strategies to manually find examples and counter-examples of UML diagrams for each extension. For each potential UML file extension, we devised heuristics based on regular expressions to tag the files that contain UML semi-automatically. After identifying UML extensions and tagging UML files, we obtain the final dataset of 552 repositories containing UML diagrams. Table 1 summarizes descriptive statistics of the projects we present as examples and case studies.

Table 1: Statistics of the Case Studies

Project	Commits	Contributors	Latest Release		
			Parsed Files	Java Entities	UML Diagrams
orekit	8,594	52	1,344	1,215	55
teammates	18,369	609	475	450	15
dataverse	27,492	179	811	806	13

4 DID DRIFT ANALYSIS

The *git* log contains information about Java and PlantUML files for each commit.[1] After extracting from the log information about all .java, .puml, and .plantuml file modifications, we employ two parsers to analyze their content in all the versions. We parse Java files with the *javalang* library.[2] To optimize the parsing we adopt a differential parsing strategy by considering only the differences between two commits. We parse PlantUML files with the *plantuml-parser*[3] to extract references to Java entities from UML diagrams.

[1]Linearized history, commits in topological order (`git log --topo-order`)
[2]See **https://github.com/c2nes/javalang**
[3]See **https://github.com/Enteee/plantuml-parser**

We end up with two instances of similar models, one for Java and one for UML, both having *packages, classes, interfaces, enums, references, methods, fields,* and *arguments*. To accommodate for UML's flexibility and get closer to the representation of Java systems, we need to disambiguate parameter types in UML method signatures, based on type separators (colon) and capitalization. We connect references in UML to entities in source code in an undirected graph (see Section 4.2). We extract information about releases[4] from GitHub, through its REST API, and add it to the cloned projects summaries.

The resulting tool is Drifter, an interactive explorer to capture DID drift in Java projects on GitHub. Drifter allows to analyze a project and select a GitHub release or a *git* commit. It presents four data visualizations, each pertaining to a different aspect of DID drift: Coverage and detailedness, relationships between UML and source code, and coverage evolution at project and file level.

4.1 Package Visualization

Figure 1 shows UML overall coverage and method-level detailedness (Section 2) of the *cs-si/orekit* GitHub project. Each innermost circle represents a Java entity (*e.g.,* class). Entities covered (*i.e.,* mentioned) by at least one UML diagram are green, those with only an implementation are white. Java entities are contained in outer circles corresponding to packages and their containment relationships.

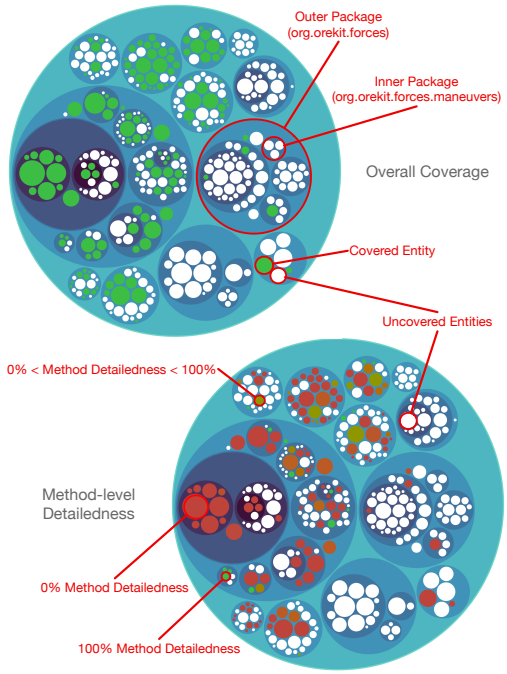

Figure 1: Package Visualization

Method-level detailedness coverage goes more in depth by considering the percentage of methods implemented in a class that is covered by the corresponding UML reference. Since multiple diagrams can contain partial references, we provide different aggregation types for detailedness metrics (*e.g.,* min, max, average).

[4]Releases correspond to *git* tags but are mined from the GitHub API separately.

Example Insights. While we can find some packages with a good overall coverage (*i.e.,* most classes are green), a low method-level detailedness of the same classes reveals that the use of UML is partial and oriented to an overview of the system architecture. When classes in some packages have 0% method-level detailedness but most of them are shown as covered in the overall coverage visualization, it likely indicates the presence of a package diagram, only mentioning the involved classes without detailing them.

4.2 UML–Source Graph

Figure 2 shows Drifter's UML–Source graph, where references in UML diagrams (green) are connected to the corresponding Java entities (blue). Graph connectivity visually represents how many diagrams a Java entity is referenced by, and how many references to Java entities are contained in each UML diagram. Undocumented Java entities are represented by unconnected nodes in the graph.

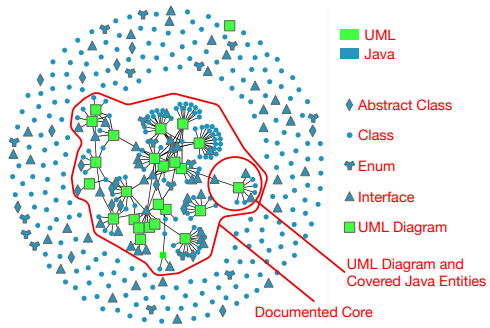

Figure 2: UML–Source Graph

4.3 Coverage History

The package and UML–Source graph visualizations are useful for exploring a specific commit of a repository, but they lack support for understanding the evolution of UML coverage, especially at a higher level. Figure 3 shows coverage history percentage over time with respect to the total number of classes in the system.

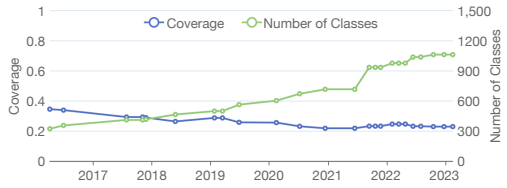

Figure 3: Coverage History Chart — Release View

Release view supports a coarse-grained representation which abstracts from single commits and low-level details of the development workflow. Commit-level coverage history is more fine-grained and can be leveraged to assess the current status and guide the development workflow and resource allocation. If, for example, the coverage drops significantly, to avoid increasing undocumented classes, existing diagrams should be updated or new ones should be created. Release view tends to be more consistent thanks to the fact that, when there is a release, the project should be in a good, stable, buildable, and therefore more reliably parsable state.

4.4 File History

Another relevant evolutionary aspect is how a single file changes over commits and releases. We compare the evolution of Java source code files with that of the UML diagrams containing a reference to the same entities (*i.e.,* the corresponding Java class).

Figure 4 shows a zoomed-in view of the file history for *Instructor.java* in the *teammates/teammates* repository. The symbols mark the commit index where each file has been added (squares), modified (circles), or removed (diamonds). A red line for Java files indicates a lack of coverage for the Java entity in the corresponding commits.

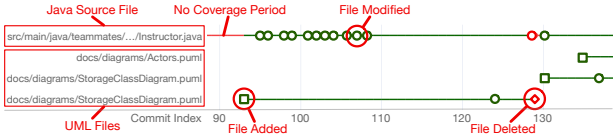

Figure 4: File History for *Instructor.java*

Example Insights. The Java source file history has a period of no coverage (red line, top-left). The *Instructor* class was not mentioned in any UML diagram until it appeared in the new file *StorageClassDiagram.puml* in commit 94 (April 2021[5]). The file was removed and added again in two subsequent commits, temporarily leaving the Instructor class uncovered. We can also notice how 11 modifications to the Instructor class after commit 96 did not have a corresponding modification to the StorageClassDiagram.puml file, indicating that either only implementation details were changed or the UML documentation could have become outdated. Finally, in commit 135, *Actors.puml* mentions the class which is therefore referenced in two different UML diagrams.

5 CASE STUDIES

We present in two case studies how we capture and understand DID drift from the interaction of multiple aspects in our characterization of the phenomenon, how our time-based approach allows identifying systems where design precedes implementation, but also contrasting wishes and reality of UML artifacts' maintenance.

Teammates. Teammates is a web-based peer feedback management system for students and teachers used by more than 800,000 users from over 1,110 universities around the world.[6] We used the Teammates project to present example insights elicited by the file history analysis (Section 4.4). Now we focus on the coverage history of Teammates in the last 2.5 years (Figure 5).

With the release 7.15, the project's documentation was migrated to PlantUML files. The chart shows an increase in coverage to 6% (22 classes out of 368) in a minor release and does not increase thereafter. In fact it reaches 5% when the system comprises up to 446 classes. In version 8.25.0, the same 22 classes are covered by 4 PlantUML diagrams, as can be confirmed by the UML–Java class graph. In the last two years, not a single class has been documented in a .puml file, despite the decision to, paraphrasing from the discussion in the migration issue, adopt PlantUML for being free and usable without the need of specialized software, version control friendly, and better supporting a more consistent use of UML notation.

[5] Drifter supports tooltip-based commit inspection (*e.g.,* date, committer, message).
[6] See **https://teammatesv4.appspot.com/web/front/home**

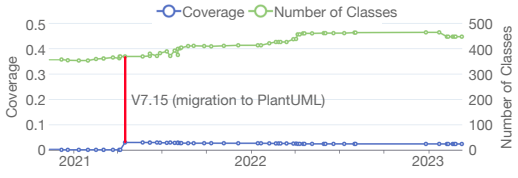

Figure 5: Teammates Release Coverage History

Similar motivations are found in another case study, *cs-si/orekit*, and are a possible reason why PlantUML is becoming popular. Nevertheless, these motivations did not correspond to an increase in the documentation effort. We also performed an analysis of the maintenance and maintainers of existing UML diagrams but this is outside the scope of the presented work.

Dataverse. Since the late adoption of PlantUML in the Teammates case study limits our analysis of the DID drift, especially in the early phases of the project where we assume a higher focus on design, we found another project that started with .puml files as its form of system documentation. Dataverse is a software platform for sharing, finding, citing, and preserving research data, and it is managed by the Institute for Quantitative Social Science (IQSS) at Harvard University. The *iqss/dataverse* project is the perfect example of "design before coding" approach.

The commit-level coverage history highlights 3 "coverage events" (spikes in coverage percentage, detailed in Figure 6). We perform an analysis of the Java files and UML Java references added in the commits constituting two of these events.

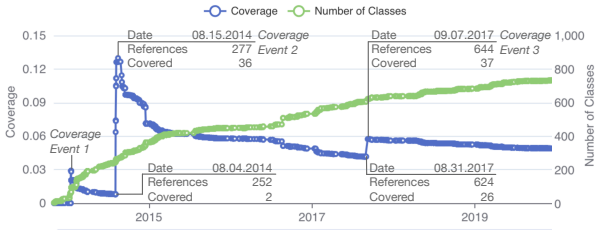

Figure 6: Dataverse Commit-Level Coverage History

Coverage event 2 sees the addition of 25 new Java classes that increase the overall coverage, indicating that those classes were already present in some UML diagrams as a design of the system that has been subsequently implemented. Nonetheless, there are 9 additions to UML diagrams contributing to the coverage increase by documenting previously undocumented classes. Coverage event 3 is more balanced in terms of design versus documentation, with 5 newly covered classes in UML diagrams and 6 new class implementations already covered by design. The Workflow class is documented and implementations of its collaborating classes WorkflowContext, WorkflowStep, WorkflowStepData, WorkflowStepResult, and WorkflowStepSPI are added to the source code.

Discussion. The case studies highlight the relevance of evolutionary analysis to assess DID drift. Manual analysis is supported by the proposed approach in highlighting points of interest in the development history. We also provide an interpretative framework to formulate hypotheses on the three-way interactions between design, implementation, and documentation leading to drift.

The example of the Workflow- classes in the Dataverse case study indicates a discussion about the design, persisted in a UML artifact, before starting the actual implementation. The procedure to identify these "*design-first*" approaches could be automated to perform a large scale study on UML practices, to empirically evaluate its usefulness and usage preferences as design or documentation tool.

6 RELATED WORK

High-quality documentation increases the success chances of a software project [11, 18, 26]. UML diagrams, although with some notable exceptions [14, 25], support developers by promoting active discussion in teams [17] and by making them achieve better functional correctness when changes can leverage accurate and up-to-date diagrams [3, 11]. We focus on human-readable text-based UML formats because of their increasing popularity in teaching and practice (for example, see Carruthers *et al.* [6] and Jasser *et al.* [16]).

Model-Driven Reverse Engineering (MDRE) [22] reconstructs model descriptions of existing systems, for example, for documentation, migration, and evolution of legacy systems (see the approach of Favre [12] or tools like MoDisco, from Brunelière *et al.* [5]). Sabir *et al.* investigated how to generate UML diagrams from existing Java source code with MDRE techniques [24]. While MDRE focuses on creating new diagrams from code, our approach leverages existing UML artifacts to analyze the status and the evolution of design, implementation, and documentation of a software project, especially in the early phases of a design-first project.

The natural divergence between architecture and implementation has been extensively studied. Architecture erosion [4, 9, 19–21], degradation [15], and consistency [2, 23] assume a design-first approach where the focus is on the effect of development on the intended architecture of the system. In our work, we show how UML can also be used as *a posteriori* documentation and we consider the role of design documents fulfilled when they become documentation for the comprehension of the implemented system.

To the best of our knowledge, our work is the first to consider a complete development iteration, from design, through implementation, to documentation, highlighting the evolution of the gap that naturally tends to affect artifacts produced in the three phases.

7 CONCLUSION

Coverage and detailedness provide information about UML as documentation but also when UML is used for design. The temporal relationships between UML and source code entities indicate whether UML is used for design or documentation. The presented project-level evolutionary analysis hints at how to make DID drift actionable to improve the quality and usefulness of UML artifacts, in turn improving system comprehension.

More needs to be done to exploit the potential of DID drift analysis. For example, evolutionary analysis to distinguish between UML for design versus UML for documentation needs to be made automatic. Support for languages other than Java and PlantUML can also improve the applicability of our approach. This work constitutes a necessary first step towards a better understanding of the interactions between UML for design and documentation, its drift from implementation, and how to mitigate it efficiently.

The authors would like to thank the Swiss Group for Original and Outside-the-box Software Engineering (CHOOSE) for sponsoring the trip to the conference.

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *Proceedings of ICSE 2019 (International Conference on Software Engineering)*. IEEE, 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[2] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. 2018. Architecture Consistency: State of the Practice, Challenges and Requirements. *Empirical Software Engineering* 23, 1 (2018), 224–258. https://doi.org/10.1007/s10664-017-9515-3

[3] Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, and Yvan Labiche. 2006. The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation. *IEEE Transactions on Software Engineering* 32, 6 (2006), 365–381. https://doi.org/10.1109/TSE.2006.59

[4] Ahmed Baabad, Hazura Binti Zulzalil, Sa'adah Hassan, and Salmi Binti Baharom. 2022. Characterizing the Architectural Erosion Metrics: A Systematic Mapping Study. *IEEE Access* 10 (2022), 22915–22940. https://doi.org/10.1109/ACCESS.2022.3150847

[5] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology* 56, 8 (2014), 1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007

[6] Sarah Carruthers, Amber Thomas, Liam Kaufman-Willis, and Aaron Wang. 2023. Growing an Accessible and Inclusive Systems Design Course with PlantUML. In *Proceedings of SIGCSE 2023 (Technical Symposium on Computer Science Education)*. ACM, 249–255. https://doi.org/10.1145/3545945.3569786

[7] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *Proceedings of MSR 2021 (International Conference on Mining Software Repositories)*. IEEE/ACM, 560–564. https://doi.org/10.1109/MSR52588.2021.00074

[8] Albert Danial. 2021. *cloc: v1.92*. Zenodo. https://doi.org/10.5281/zenodo.5760077

[9] Lakshitha de Silva and Dharini Balasubramaniam. 2012. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software* 85, 1 (2012), 132–151. https://doi.org/10.1016/j.jss.2011.07.036

[10] Stephane Ducasse and Damien Pollet. 2009. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering* 35, 4 (2009), 573–591. https://doi.org/10.1109/TSE.2009.19

[11] Wojciech J. Dzidek, Erik Arisholm, and Lionel C. Briand. 2008. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Transactions on Software Engineering* 34, 3 (2008), 407–432. https://doi.org/10.1109/TSE.2008.15

[12] Liliana Favre. 2008. Formalizing MDA-Based Reverse Engineering Processes. In *Proceedings of SERA 2008 (International Conference on Software Engineering Research, Management and Applications)*. IEEE, 153–160. https://doi.org/10.1109/SERA.2008.21

[13] Martin Fowler. 2018. *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley.

[14] Carmine Gravino, Giuseppe Scanniello, and Genoveffa Tortora. 2015. Source-Code Comprehension Tasks Supported by UML Design Models: Results From a Controlled Experiment and a Differentiated Replication. *Journal of Visual Languages and Computing* 28 (2015), 23–38. https://doi.org/10.1016/j.jvlc.2014.12.004

[15] Sebastian Herold, Martin Blom, and Jim Buckley. 2016. Evidence in Architecture Degradation and Consistency Checking Research: Preliminary Results from a Literature Review. In *Proccedings of ECSA 2016 (European Conference on Software Architecture Workshops)*. ACM, Article 20, 7 pages. https://doi.org/10.1145/2993412.3003396

[16] Muhammed Basheer Jasser, Lee Ming Zhen, Bayan Issa, Ling Mee Hong, and Ismail Ahmed Al-Qasem Al-Hadi. 2023. Quantifying Object-Oriented System Complexity: Introducing a Powerful Measurement Tool. In *Proceedings of ICSET 2023 (International Conference on System Engineering and Technology)*. IEEE, 221–226. https://doi.org/10.1109/ICSET59111.2023.10295081

[17] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Juraj Vincur, Regina Hebig, Xavier Le Pallec, Michel Chaudron, Sébastien Gérard, Ivan Polasek, and Andreas Wortmann. 2022. The Influence of Software Design Representation on the Design Communication of Teams with Diverse Personalities. In *Proceedings of MODELS 2022 (International Conference on Model Driven Engineering Languages and Systems)*. ACM, 255–265. https://doi.org/10.1145/3550355.3552398

[18] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software* 20, 6 (2003), 35–39.

[19] Ruiyin Li, Peng Liang, Mohamed Soliman, and Paris Avgeriou. 2021. Understanding Architecture Erosion: The Practitioners' Perceptive. In *Proceedings of ICPC 2021 (International Conference on Program Comprehension)*. IEEE, 311–322. https://doi.org/10.1109/ICPC52881.2021.00037

[20] Ruiyin Li, Peng Liang, Mohamed Soliman, and Paris Avgeriou. 2022. Understanding Software Architecture Erosion: A Systematic Mapping Study. *Journal of Software: Evolution and Process* 34, 3 (2022), e2423. https://doi.org/10.1002/smr.2423

[21] Ruiyin Li, Mohamed Soliman, Peng Liang, and Paris Avgeriou. 2022. Symptoms of Architecture Erosion in Code Reviews: A Study of Two OpenStack Projects. In *Proceedings of ICSA 2022 (International Conference on Software Architecture)*. IEEE, 24–35. https://doi.org/10.1109/ICSA53651.2022.00011

[22] Claudia Raibulet, Francesca Arcelli Fontana, and Marco Zanoni. 2017. Model-Driven Reverse Engineering Approaches: A Systematic Literature Review. *IEEE Access* 5 (2017), 14516–14542. https://doi.org/10.1109/ACCESS.2017.2733518

[23] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. 2011. Assessing Architectural Drift in Commercial Software Development: A Case Study. *Software: Practice and Experience* 41, 1 (2011), 63–86. https://doi.org/10.1002/spe.999

[24] Umair Sabir, Farooque Azam, Sami Ul Haq, Muhammad Waseem Anwar, Wasi Haider Butt, and Anam Amjad. 2019. A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code. *IEEE Access* 7 (2019), 158931–158950. https://doi.org/10.1109/ACCESS.2019.2950884

[25] Giuseppe Scanniello, Carmine Gravino, Marcela Genero, Jose' A. Cruz-Lemus, and Genoveffa Tortora. 2014. On the Impact of UML Analysis Models on Source-Code Comprehensibility and Modifiability. *ACM Transactions on Software Engineering and Methodology* 23, 2, Article 13 (2014), 26 pages. https://doi.org/10.1145/2491912

[26] Eirik Tryggeseth. 1997. Report from an Experiment: Impact of Documentation on Maintenance. *Empirical Software Engineering* 2, 2 (1997), 201–207. https://doi.org/10.1109/MS.2003.1241364

Received 22 Nov 2023; revised 28 Jan 2024; accepted 10 Jan 2024