Mining the Relationship between Object-Relational Mapping Performance Anti-patterns and Code Clones

Zeshan Xu[†‡], Jie Zhu[†‡], Li Yang[*†], Chun Zuo[§]
[†]Institute of Software, Chinese Academy of Sciences, Beijing, China
[‡]University of Chinese Academy of Sciences, Beijing, China
[§]Sinosoft Co.,Ltd., Beijing, China
{xuzeshan21, zhujie212}@mails.ucas.ac.cn
yangli2017@iscas.ac.cn
zuochun@sinosoft.com.cn

***Abstract*—The use of Object-Relational Mapping (ORM) in software development has become increasingly popular due to its superiority to simplify database interactions. Despite the prosperous development of ORM code smells detection tools for general code smell problems related to coupling and cohesion, these tools do not capture issues that are specific to ORM code statement context. In this work, we fill the gap wit h the potential performance anti-patterns of repetitive ORM code by heuristic analysis and code clone analysis on 6 open source ORM systems in Java and Python (Saler, Wagtail, Zulip, Taiga, Protal, and Roller). For each occurrence of this code smell, we distinguished problematic instances that potentially require further fixes among justifiable ones. Through our research, we identified four anti-patterns associated with repetitive ORM code and proposed fix strategy for each of these anti-patterns. Additionally, our study delves into the relationship between repetitive ORM code anti-patterns and code clone, which reveals that a substantial proportion of repetitive ORM statements can be found in cloned code. Experiments show that repetitive ORM code can lead to a waste of system performance. This research highlights the impact of ORM code context on the proper use of ORM frameworks and emphasizes that copying ORM code without context evaluation can be detrimental to system performance.**

***Index Terms*—code clone, anti-pattern, code smell, object-relational mapping, static analysis, empirical study**

I. Introduction

As a well-established programming technique, Object-Relational Mapping (ORM) has emerged as a solution to the problem of Object-Relational Impedance Mismatch [1], practitioners employ ORM frameworks to close the divide between databases and applications by handling the tasks of data mapping and persistence [2], [3]. However, traditional static code analyzers are insufficient in capturing the unique attributes of ORM code and thus unable to identify potential issues within ORM code [4]. Rahman et al. reported that 76 of 77 projects don't follow the rule recommendations of Hibernate architecture [5]. Rahman et al's latest study identified and defined a greater number of code smells within the field of ORM [6]. However, previous approaches are still coarse-grained and only consider the appropriateness of ORM statement only, while operations involving ORM persistent objects, such as querying, saving, modifying, and deleting, all entail contextual information [3]. In other words, we consider that utilizing ORM statements requires taking into account the contextual information of code. Although each ORM statement may be innocuous on its own, duplicating it without considering the context is likely to result in unnecessary performance overhead of the new context. Fig 1 shows an example of repetitive ORM code from Saler. The developer defined the exact same ORM statement in both interfaces, but used different interface names. findByUsername sounds from the name to find a single piece of data through Username, while findAllByUsername finds all data through Username. When the developer only wants to obtain a piece of data, using the findByUsername interface will lead to a lot of unnecessary query data, resulting in a waste of ORM system performance.

Instances of Raw ORM Code

```
public List<Bonus> findAllByUsername(String username) throws DataAccessException {
...
return this.entityManager.createQuery(
"SELECT o FROM Bonus o WHERE o.user.username = :username ORDER BY o.id DESC")
...
}
```

```
public List<Bonus> findByUsername(String username) throws DataAccessException {
...
return this.entityManager.createQuery(
"SELECT o FROM Bonus o WHERE o.user.username = :username ORDER BY o.id DESC")
...
}
```

Fig. 1. A repetitive ORM code example. The ORM framework part of the code is marked in blue, and the differences are marked in red.

To help developers improve ORM practices, in this paper, we focus on studying repetitive ORM statements. We conducted heuristic analysis on 6 open source ORM systems written in Java and Python. Our investigation revealed four repetitive ORM code anti-patterns.

Intuitively, repetitive ORM statements could be related to,

*Corresponding author.
DOI reference number: 10.18293/SEKE2023-161

or are even a consequence of code clones. During software development, developers are often tasked with performing unfamiliar programming tasks [7]. When confronted with these challenges, developers frequently resort to online searches or scouring through project code. In the search results, code snippets are the ideal resources for developers to leverage during the development process [8], especially for unfamiliar tasks. A code snippet refers to a segment of code that achieves one or multiple specific programming objectives [9]. Code snippets can be directly reused by copy-pasting [10], which may contain repetitive ORM code statements.

Fig 2 presents the overall framework of our research. We examined 177 open source ORM systems written in Java and Python, using a Criticality Score [10] metric to guide our choices, and performed detailed heuristic inspections on six of them (Saler, Wagtail, Zulip, Taiga, Protal, and Roller) to identify recurring ORM anti-patterns. To further investigate the relationship between repetitive ORM fragments and code clone, we used NiCad [5], a code clone detection tool, to analyze each revision of these 6 projects. Combined with our heuristic analysis, we found that code clone resulted in a large number of repetitive ORM fragments. These results suggest that code clone may be a significant contributing factor to the presence of repetitive ORM fragments in software systems.

Specifically, we make the following contributions:

- We uncovered four new ORM performance anti-patterns through a comprehensive heuristic analysis of over 1K instances of repeated ORM code statements in six open-source ORM systems with suggested fixes for each anti-pattern.
- We found that code clone can lead to performance anti-patterns in ORM systems, and the majority of problematic repetitive ORM code anti-patterns (90.2%) were found to exist in cloned code fragments.
- We discovered that 73% of the instances of the repetitive ORM code anti-patterns that were not detected by code clone detection tools were actually from microcloned fragments. This finding highlights the need for future studies to investigate the negative impact of microclone on system performance.
- We found that reckless code clone can lead to performance anti-patterns for ORM frameworks and discovered 153 ORM performance anti-patterns in six large open source systems.

All data are publicly available[1].

II. Related Work

A. Empirical Studies on ORM System

Several studies have investigated Object-Relational Mapping (ORM) systems [4], [11], with Chen et al. [4] examining open source ORM systems and highlighting the hidden maintenance costs of using ORM frameworks. Meanwhile, SAddAR et al. [11] conducted empirical research on the performance of ORM frameworks. These studies have shown that although

[1]https://figshare.com/s/e5a20d2267b08a39018e

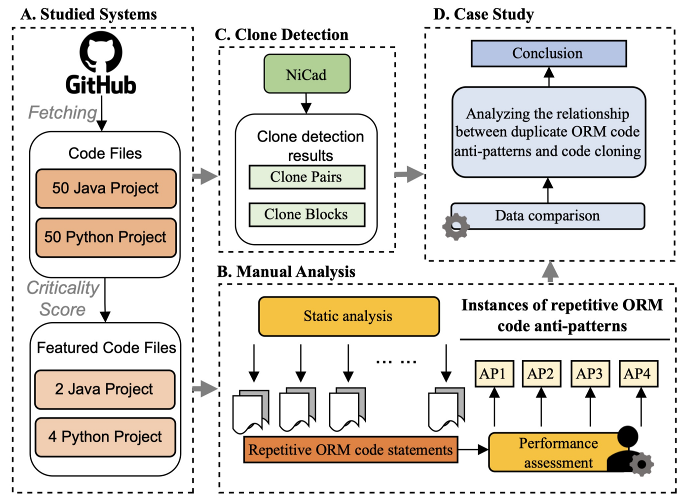

Fig. 2. The overall process of our study. AP refers to four repetitive ORM code anti-patterns.

there are several benefits of using an ORM framework, maintaining ORM code can pose certain challenges. Specifically, Chen et al. [4] found that ORM code undergoes more frequent changes than regular code and lacks automated verification and detection methods, based on their analysis of different revisions of open source systems. In contrast, our study focuses on repeated ORM statements and code clone in the system, incorporating context (i.e., surrounding code) to find and identify methods for addressing ORM code issues.

B. Code Smells and Anti-patterns

Code smells and anti-patterns can be indicators of poor design and implementation of code, which can adversely affect the maintainability [11]–[14], understandability [15], [16], and performance [17] of a software system. To address their effects, several studies have proposed detecting code smells and anti-patterns [18]–[20]. In the context of the ORM system that we investigated, Holder et al. [21] proposed a metric suite to measure the complexitAy of ORM mapping code, while Silva et al. [5] suggested a set of rules to verify whether Hibernate entity code follows the JPA [22] specification. Loli et al. [23] surveyed previous research [24]–[26] and proposed ORM code smells in the literature, which surveyed developers' agreement on the definition of smell. The findings indicated that most developers agreed with the definition and severity of the smells.

Code clone, or repetitive code, is a code smell that can arise when a developer copies and pastes a piece of code from one place to another [27]. This code clone method may lead to software quality problems [28]–[30], and there are some works in previous research committed to detect and solve these odors through various methods [5], [6], [16], [31].

III. Methodology

In this section, we will describe our methodology, which includes two parts: (1) How we identify repetitive ORM

statements in existing open source ORM systems for heuristic research and (2) How we conduct heuristic research to investigate which code repetitive ORM statements have anti-patterns.

A. Studied Systems

We manually analyzed six of open source ORM system in table I. To generate a practical dataset, We combined the dataset of 77 projects mentioned in previous research on ORM systems [5], [6] with 100 open source projects using ORM on github. we used the open source project importance score[2] to screen and evaluate projects. To eliminate the relationship between the ORM framework and the programming language, we collected not only Java projects but also Python projects, primarily those using the Django and Hibernate framework.

B. Define Repetitive ORM Statements

In this paper, we define repeated ORM statements as statements with the same conditional call structure within the ORM framework. For instance, we consider the two ORM statements below to be duplicates:

$$
\begin{aligned}
&Product.objects.filter(vector = None).\\
&\quad prefetch_related(\text{“}data\text{”}).order_by()\\
&Product.objects.filter(document = \text{“”}).\\
&\quad prefetch_related(\text{“}addresses\text{”}).order_by()
\end{aligned}
$$

C. Identify Repetitive ORM Statements.

We employed static analysis to analyze the source code. We defined ORM statements as an abstract combination of models, methods, and parameters, which can exhibit various forms of queries, additions, deletions, and modifications based on their intended purposes. We excluded repetitive ORM statements used for model building and extracted parameter information, such as table and column names, to facilitate heuristic analysis.

IV. Anti-patterns Of Repetitive ORM Code

Similar to previous research on anti-patterns, we consider repetitive ORM anti-patterns as "superficial indicators of deeper problems in the system" [32]. The presence of repeated ORM code may indicate an underlying problem that requires attention. However, not all instances of repetitive ORM code are problematic. We categorized each repetitive ORM statement as either problematic or reasonable to ignore based on the context of the repetitive ORM code (i.e., the surrounding code). Our research can help developers improve their ORM coding practices and inspire future research in this area.

In total, we examined 393 groups of repetitive ORM statement pairs, constituting more than 1K ORM statements in aggregate. Each group comprised two or more ORM statements with matching conditional call structures. In total, we discovered 153 problematic instances in the ORM code, featuring four recurring anti-patterns: 1) Selecting superfluous data (Select All), 2) Retrieving related data that remains unused (Associated Object), 3) Employing redundant sorting procedures (Order Waste), and 4) Repeatedly fetching non-updated data (Cache Waste). It is noteworthy that a single set of repetitive ORM code pairs could encompass multiple issues. Specifically, we observed these recurring anti-patterns occurring 138, 26, 14, and 10 times, respectively.

[2]https://github.com/ossf/criticality_score

Table II displays the number of instances of each anti-pattern that we identified manually. We will provide a comprehensive discussion of each anti-pattern and propose suitable solutions below.

Select all. In our investigation, we discovered that when a repeated ORM query statement selects entity objects from the DBMS, the original query selects all columns of an object, while the cloned repeated ORM statement only uses a small number of columns. For instance, let us consider the Proxy class. If the code cannot use all the columns of the Proxy, it is recommended to add filter column conditions before the query statement. As the ORM framework lacks knowledge about the required data, it selects all columns by default, leading to avoidable performance degradation. **Fix Strategy:** Customize the specific parameters of the ORM query based on the context, and include the appropriate column tag in the statement.

$$
\begin{aligned}
&A = ProxyModel.objects.filter(\\
&\quad cluster_name = \text{“}A\text{”}, dc = \text{“}lf\text{”}, service = \text{“}nsq\text{”})\\
&ip, port = A.ip, A.port\\
&A = ProxyModel.objects.filter(\\
&\quad cluster_name = \text{“}A\text{”}, dc = \text{“}lf\text{”}, service = \text{“}nsq\text{”})\\
&ip = A.ip
\end{aligned}
$$

Associated object. This anti-pattern indicates that associated object data has been queried unnecessarily. The problem with this anti-pattern is that it retrieves unnecessary tuples from other database tables. When using the ORM framework, developers can specify the relationship between entity classes, such as one-to-one, one-to-many, many-to-one, and many-to-many. The ORM framework provides different optimization techniques to specify how to obtain associated entity objects from the database. For example, in the code, if only the Users information is needed, but the ORM framework retrieves the associated Books data, then it causes unnecessary overhead in terms of performance. Retrieving users together with associated objects can be expensive, particularly when there are many Books in Users. Previous studies have shown that ORM frameworks often use SQL connections to obtain too much data, which can significantly reduce system performance [33]. Different ORM frameworks provide different methods to solve this redundant data problem, and our approach can provide developers with guidance to address such issues. **Fix Strategy:** Instead of directly utilizing the duplicated old model, establish a new model to access the necessary table.

Sort waste. This anti-pattern involves performing unnecessary sorting operations. Sorting the queried data is a commonly used technique by developers in ORM. There are two ways to

TABLE I
PROJECTS IN OUR RESEARCH

System	About	Total lines of code	Language	Stars	Commits	Version
Saler	Saleor Core: The high performance, composable, headless commerce API	435882	Python	18.2k	20284	3.12.3
Wagtail	A Django content management system focused on flexibility and user experience	198763	Python	13.4k	14805	4.2.1
Zulip	Open-source team chat that helps teams stay productive and focused	258309	Python	17.4k	50313	6.1.0
Taiga	Agile project management platform. Built on top of Django and AngularJS	113156	Python	5.8k	4164	1.0.0
Protal	Devproof Portal - Available modules Blog, Articles, Downloads, Boomarks	53725	Java	14	971	1.0.0
Roller	Java-based open-source blog server that uses Hibernate for database interactions	96954	Java	116	4722	1.0.0

sort data in ORM: using the order_by() method or specifying the sort order when defining the model. The default is ascending order, but it can be changed to descending order using the desc() method. And the column name for sorting could also be customized. Studies have shown that sorting operations can significantly degrade SQL performance [41], particularly when sorting tables that do not contain indexes. To further investigate this issue, we analyzed the time consumption of ORM statements with and without sorting operations using the database Explain function (i.e., the execution plan). Our results showed that redundant sorting operations accounted for 47% of performance waste. This finding highlights the importance of optimizing sorting operations in ORM coding practices. **Fix Strategy:** Evaluate the need for sorting based on the context, and remove unnecessary sorting actions.

Cache waste. Our investigation found that ORM frameworks offer support for caching [34], which involves sharing objects between transactions to improve performance. However, striking a balance between performance and data staleness can be challenging in distributed systems that use caching. For instance, consider the following example:

$$RegionModel.objects.filter(id = 481).$$
$$update(\text{“}region\text{”} = \text{“}cn\text{”})$$
...
$$RegionModel.objects.filter(id = 481).$$
$$values_list(\text{“}region\text{”})$$
...
$$RegionModel.objects.filter(id = 481).$$
$$values_list(\text{“}region\text{”})$$

The first query statement is necessary because the user data has been updated by another query for the same primary key in the filter clause. In contrast, the second query does not need a second query since the data has not changed. While most ORM frameworks provide caching mechanisms to reuse fetched data and minimize database access, the caching configuration is not automatically optimized for different application systems [34], and some ORM frameworks even disable caching by default. Even if the retrieved entity object has not been modified, this repeated ORM statement may execute millions of times in a short period. Although different ORM frameworks may have distinct configurations, the issues we report are common, and our finding can aid developers in optimizing caching. **Fix Strategy:** Remove the redundant ORM code that does not modify the data, and utilize the same object generated by the context for accessing purposes.

TABLE II
NUMBER OF PROBLEMATIC INSTANCES FOUND BY OUR RESEARCH

System	Select All	Associated Object	Order Waste	Cache Waste
Saler	62	6	8	2
Wagtail	13	8	2	0
Zulip	32	3	0	4
Taiga	15	5	0	3
Protal	4	0	1	1
Roller	12	4	3	0
Total	138	26	14	10

In our investigation, we identified 153 instances of ORM code anti-patterns in 1301 cloned blocks. Our findings suggest that reckless code clones can result in performance anti-patterns for ORM frameworks. This highlights the importance of identifying and addressing such anti-patterns to ensure optimal performance of ORM systems.

V. RELATION BETWEEN REPETITIVE ORM CODE ANTI-PATTERNS AND CODE CLONE

A. Motivation

Code clone, or duplicating code, is a development pattern that is generally considered harmful to software maintainability, understandability, and performance. Previous research has focused on studying code clones in source code and understanding their impact. However, since cloning is often done in a hurry and without paying much attention to code context, ORM-related code may be copied as well. In the previous section, we identified four performance anti-patterns (i.e., Select All, Associated Object, Order Waste and Cache Waste) that can result from repetitive ORM code. However, not all repetitive ORM code is necessarily the result of code clone. In this section, we investigate the Relationship between Object-Relational mapping Performance anti-patterns and code clones. By doing so, we hope to provide developers with better practices for ORM code and encourage further research on code clone in this context.

B. Method

We use NiCad to detect clones, which has high precision (95%) and recall (96% identical). NiCad detects all major types of clones, including exact (type 1) and near-miss (types 2 and 3) clones, and is actively maintained (with the latest version released in November 2020). Clone detection results may vary based on different detector settings, so choosing appropriate parameters is important. In our research, we set the granularity of source code units to block level and used NiCad to detect block clones with a minimum size of 10 LOC and a similarity threshold of 70%, as recommended in a previous study [35]. This approach provided better clone detection results in terms of precision and recall.

We utilized NiCad to perform experiments on the open-source systems of the six studies mentioned earlier. Subsequently, we analyzed the clone detection results and compare the locations of clones with those of problematic instances. If two or more cloned snippets contained the same set of instances, we regarded those instances as related to the clone. Through manual screening and analysis, we classified the results of code clone into two categories: clone blocks that involved database calls and those that did not. To mitigate the impact of false negatives, we conducted additional heuristic research on all instances that NiCad failed to identify as clones. Our focus was on blocks of code that surrounded ORM statements and exceeded a threshold of 10 lines, which was the same as that of the clone detection tool.

C. Result

TABLE III
NUMBER OF PROBLEMATIC INSTANCES OF REPETITIVE ORM CODE ANTI-PATTERN DETECTED AS CLONES BY NICAD

System	Clone Blocks	Select All	Associated Object	Order Waste	Cache Waste
Saler	566	62/62	5/6	3/8	1/2
Wagtail	232	13/13	8/8	2/2	0
Zulip	61	32/32	3/3	0	0/4
Taiga	209	15/15	3/5	0	0/3
Protal	92	4/4	0	1/1	0/1
Roller	141	12/12	0/4	3/3	0
Total	1301	138/138	19/26	11/14	0/10

As presented in table III, we identified a total of 138 problematic occurrences of repetitive ORM anti-patterns. In comparison to the 153 instances of recurring ORM anti-patterns discovered through heuristic analysis, it can be inferred that 90% of the replicated ORM code stemmed from code clone. Specifically, our results suggest that code clone might be the primary cause of repetitive ORM code. Notably, 100% of Select All anti-patterns were detected in code clone, while only one instance of Cache Waste was found in code clone. To investigate this phenomenon further, we noted that it may be related to the threshold (line 10) established for code clone detection. The repetitive ORM statements belonged to micro-codes comprising less than ten lines of code, and prior research has indicated that microclone is also crucial for consistency in updates [36], but they are challenging to detect due to their limited size.

> We found that code clone can lead to performance anti-patterns in ORM systems, and the majority of problematic repetitive ORM code anti-patterns (90.2%) were found to exist in cloned code fragments.

Results revealed that 10% (15/153) of problematic occurrences of repetitive ORM code anti-patterns were identified as non-clones by automated code clone detection tools. To delve deeper into this issue, we scrutinized each undetected instance and found that 73% (11/15) were, in fact, derived from micro-cloned fragments. This observation underscores the need for future research to explore the detrimental effects of microclone on system performance and to develop guidelines for writing ORM code correctly. Our findings justify the need for future research to investigate the negative impact of microclone on system performance. In addition to considering code maintenance and refactoring overhead, future studies on code clone detection should also consider other possible side effects of code clone. Researchers can also optimize code clone detection methods by refining the detection of various types of statements commonly found in ORM code to improve its accuracy.

> We found that 73% of instances of repetitive ORM code anti-patterns that were not detected by code clone detection tools were actually from microcloned fragments.

VI. CONCLUSION

ORM are widely used to solve the object-relational impedance mismatch problem by providing an object-oriented interface on top of a relational database, which enables easy saving and retrieval of program objects from secondary storage without mapping application data to database records. However, the quality of ORM code is often criticized by developers, especially its data persistence and query code. This paper presents an empirical study that investigates anti-patterns caused by duplication of ORM code (or code clone) in ORM systems. The study analyzes six open-source ORM systems (Saler, Wagtail, Zulip, Taiga, Portal, and Roller), identifies four new ORM code anti-patterns and proposed fix strategy for each of these anti-patterns, showing that repetitive ORM code can lead to a waste of system performance. These findings could serve as a valuable reference and provide direction for future research on ORM systems and optimization of ORM development practice standards.

The study also investigates the relationship between the repetitive ORM code anti-pattern and code clone. The results indicate that most problematic instances of repetitive ORM code occur in code clones, which are more likely to be microclones that are difficult to detect using existing code clone detection tools. This research highlights the impact of ORM code context on the proper use of ORM frameworks, emphasizing that copying ORM code without context evaluation can be detrimental. Therefore, future research should

consider code context when providing guidance for ORM practices, and richer contextual information is required to construct new excellent ORM research data.

VII. Acknowledgements

We sincerely appreciate the valuable feedback from the anonymous reviewers.This work was supported by the Chinese Science and Technology Aid Project to Developing Countries (KY201906007) and the National Key R&D Program of China (No.2021YFC3340204).

References

[1] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *information and software technology*, vol. 82, pp. 1–18, 2017.

[2] G. Vial, "Lessons in persisting object data using object-relational mapping," *IEEE software*, vol. 36, no. 6, pp. 43–52, 2018.

[3] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *information and software technology*, vol. 82, pp. 1–18, 2017.

[4] T.-H. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An empirical study on the practice of maintaining object-relational mapping code in java systems," in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 165–176, 2016.

[5] T. M. Silva, D. Serey, J. Figueiredo, and J. Brunet, "Automated design tests to check hibernate design recommendations," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pp. 94–103, 2019.

[6] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "Hbsniff: A static analysis tool for java hibernate object-relational mapping code smell detection," *Science of Computer Programming*, vol. 217, p. 102778, 2022.

[7] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, "Rosf: Leveraging information retrieval and supervised learning for recommending code snippets," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 34–46, 2016.

[8] L. Ai, Z. Huang, W. Li, Y. Zhou, and Y. Yu, "Sensory: leveraging code statement sequence information for code snippets recommendation," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 27–36, IEEE, 2019.

[9] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 664–675, 2014.

[10] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 260–270, IEEE, 2015.

[11] T.-H. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An empirical study on the practice of maintaining object-relational mapping code in java systems," in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 165–176, 2016.

[12] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 58–67, IEEE, 2017.

[13] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.

[14] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pp. 225–234, 2016.

[15] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 405–416, IEEE, 2017.

[16] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pp. 125–134, Ieee, 2011.

[17] X. Xiao, S. Han, C. Zhang, and D. Zhang, "Uncovering javascript performance code smells relevant to type mutations," in *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30-December 2, 2015, Proceedings 13*, pp. 335–355, Springer, 2015.

[18] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268–278, IEEE, 2013.

[19] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 282–285, 2012.

[20] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, pp. 549–575, 2015.

[21] S. Holder, J. Buchan, and S. G. MacDonell, "Towards a metrics suite for object-relational mappings," in *Model-Based Software and Data Integration: First International Workshop, MBSDI 2008, Berlin, Germany, April 1-3, 2008. Proceedings*, pp. 43–54, Springer, 2008.

[22] S. P. R. Katamreddy and S. S. Upadhyayula, "Working with jdbc," in *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, pp. 101–118, Springer, 2022.

[23] S. Loli, L. Teixeira, and B. Cartaxo, "A catalog of object-relational mapping code smells for java," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pp. 82–91, 2020.

[24] T.-H. Chen, "Improving the quality of large-scale database-centric software systems by analyzing database access code," in *2015 31st IEEE International Conference on Data Engineering Workshops*, pp. 245–249, IEEE, 2015.

[25] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, 2016.

[26] P. Węgrzynowicz, "Performance antipatterns of one to many association in hibernate," in *2013 Federated Conference on Computer Science and Information Systems*, pp. 1475–1481, IEEE, 2013.

[27] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," *Empirical Software Engineering*, vol. 17, pp. 503–530, 2012.

[28] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 485–495, IEEE, 2009.

[29] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008.

[30] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 311–320, 2011.

[31] T. Sotiropoulos, S. Chaliasos, V. Atlidakis, D. Mitropoulos, and D. Spinellis, "Data-oriented differential testing of object-relational mapping systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1535–1547, IEEE, 2021.

[32] M. A. Saca, "Refactoring improving the design of existing code," in *2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)*, pp. 1–3, IEEE, 2017.

[33] J. Dubois, "Improving the performance of the spring-petclinic sample application," 2013.

[34] M. Keith and R. Stafford, "Exposing the orm cache: Familiarity with orm caching issues can help prevent performance problems and bugs.," *Queue*, vol. 6, no. 3, pp. 38–47, 2008.

[35] M. Mondal, C. K. Roy, and K. A. Schneider, "Spcp-miner: A tool for mining code clones that are important for refactoring or tracking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 484–488, IEEE, 2015.

[36] M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 50–60, IEEE, 2018.