A Catalog of Object-Relational Mapping Code Smells for Java

Samuel Loli
Federal University of Pernambuco
Recife, Brazil
sbl@cin.ufpe.br

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

Bruno Cartaxo
Federal Institute of Pernambuco
Paulista, Brazil
email@brunocartaxo.com

ABSTRACT

Bad choices during software development might lead to maintenance and performance issues. Code smells are typically used to indicate such problems. A number of smells have been proposed, usually focused on generic code problems. In this work, we focus on the specifics of Object-Relational Mapping (ORM) code in Java. Developers use ORM frameworks to abstract the complexity of accessing a database. However, when poorly used, frameworks can lead to problems that might affect the overall performance of the system. Therefore, we present a catalog of eight smells extracted from the state of research and practice, through a combination of rapid review and grey literature review. For each smell, we also present a suggested solution and rationale. To evaluate the catalog, we conducted a survey with 86 respondents. The majority of the respondents agree both that the code smells are a problem, as well as that the suggested solution is adequate. In conclusion, this work contributes with a systematic way of describing ORM code smells and an initial catalog, which can be useful for researchers and practitioners, positively evaluated by our initial results.

ACM Reference Format:
Samuel Loli, Leopoldo Teixeira, and Bruno Cartaxo. 2020. A Catalog of Object-Relational Mapping Code Smells for Java. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422432

1 INTRODUCTION

When developing software, incorrect choices during the design phase can lead to maintenance and performance problems. According to Fowler [15], less-than-ideal implementation symptoms that indicate low quality code are called *code smells*. In the literature, previous studies show that code smells affect code maintainability [50] and also indicate potential for future refactorings [27]. Although this definition is quite general and seems to suggest several type of code smells that may affect systems, these are typically focused on code, without considering domain-specific characteristics of the systems and frameworks used during development.

ORM frameworks are popularly used to ease data management in software development. They allow conceptual abstraction between object-oriented systems and data stored in relational databases [9]. However, this abstraction may lead to risks. According to Chen et al. [8], developers often create ORM-specific code focusing on producing code that compiles, without taking into account relevant aspects such as database performance or code maintainability. This can cause problems, such as recovering excessive data from database, wasting computational resources, or performing N queries when only one would be enough, which might lead to transaction timeouts or even large-scale system failures.

In their study on ORM code maintainability in Java, Chen et al. [10] conclude that changes to such code are complex, and future studies should study the root causes of the problems to help developers better design ORM code. In this context, this research proposes a catalog with eight ORM-specific code smells in Java, together with suggested solutions. The catalog was developed from the study of problems, causes and solutions regarding Java ORM frameworks, extracted from the state of research and practice, through a combination of rapid review [5] and grey literature review [17]. Literature reviews are a widely recognized and accepted way to synthesize knowledge spread into various sources, structuring it in an accessible way to benefit the scientific and professional community [45].

The proposed catalog is evaluated through a survey with developers that are familiar with Java ORM frameworks. They discussed the extent to which they agree that the smells are actual problems, and that the suggested solutions are indeed candidates to fix those problems. We received 86 responses, with an average agreement of 75.44% between smells and solutions presented in the catalog.

The remainder of this paper is organized as follows. Section 2 illustrates potential problems caused by ORM code smells. Section 3 describes the research methodology used to develop the catalog presented in Section 4. Section 5 presents the catalog evaluation. We discuss threats to validity in Section 6, and present the related work in Section 7. Finally, we conclude in Section 8.

2 MOTIVATING EXAMPLE

In this section, we present a motivating example adapted from an existing Java system — in which one of the authors works on, as developer — to illustrate problems arising from ORM-related code smells. ORM frameworks provide developers with a conceptual abstraction by mapping database records to objects [4, 25]. Java ORM frameworks use the Java Persistence API (JPA) standard [13, 26] to associate classes, attributes, associations, and types to the corresponding tables, columns, relationships and SQL types, respectively. This association occurs through Java annotations or XML. In Example 1, we see that the `@Entity` annotation is used to identify the `Person` class as an entity in the database, together with `@Table` to identify the table name. The `@Id` annotation defines the attribute corresponding to the primary key. We can map attributes to different column names using `@Column`, while `@ManyToOne` identifies many-to-one relationships, as seen with `Gender` and `Address` entities. There is also an implicit relationship between `Address`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBES '20, October 21–23, 2020, Natal, Brazil
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422432

and `City`, omitted from the example. The `main` method then retrieves a list of `Person` whose `name` start with "Jh", by calling the `findPersons` method, which performs an ORM query using the Java Persistence Query Language (JPQL) provided by JPA [26].

The ORM framework generates SQL queries based on the code, which might present problems, such as retrieving excessive data from the database. The goal of this particular code snippet is to get a list of full names, but the resulting queries retrieve all columns in the `Person`, Gender, `Address` and `City` tables. It would be enough to retrieve the `name` column from the `Person` table. Retrieving excessive data from the database can be considered the major performance problem in most applications that use JPA [35]. Another problem is the unnecessary queries related to the `Address` and `Gender` tables. This causes the $N + 1$ problem, where a single ORM query leads to N others, according to the number of records in the database, which increases the severity of the problem over time.

These problems could be detected by a domain-specific smell detection tool. Fowler [15] defines code smells as symptoms of poor design and implementation choices. In the presented example there are three ORM-specific code smells: (i) Eager as a Fetch Strategy in Class-level (static) Relationships; (ii) Lack of Join Fetch in ORM Queries to Retrieve Objects With Eager Attributes; and (iii) Data Retrieval Without Projection for Read-Only.

3 RESEARCH METHODOLOGY

Based on problems such as the reported in the previous section, we set out to define a catalog of ORM-specific code smells. To do so, we performed a Rapid Review (RR) combined with a Grey Literature Review (GLR). Literature reviews are a widely recognized and accepted way to synthesize knowledge spread into various sources, structuring it in an accessible way to benefit the scientific and professional community [45]. In addition, they also can help to achieve non-trivial conclusions, which might not be possible with isolated studies. Our catalog presents all smells in one place in a structured way, also adding possible suggestions to fix them. Analyzing the evidence resulting from both techniques helped us on establishing the catalog of ORM code smells in Java. The following sections provide further details over our methodology.

3.1 Method

We followed the procedure described by Cartaxo et al. [5] to conduct the RR. They define RRs as secondary studies based on adaptations of regular systematic reviews. Adaptations are performed to attend to restrictions of professionals, such as time and costs, and used to obtain a connection with the state of practice in software engineering. The use of RR has the main objective of providing evidence to assist decision making regarding problems that professionals usually have. Thus, RR is the best fit in our context, since this research was initially motivated by a problem in a real-world project.

Due to the practical aspect of the research, we also used grey literature, as recomended by Garousi et al. [17]. GLR is a particular type of Systematic Literature Review (SLR) which allows including grey literature as primary source. It might include evidence such as white papers, blogs, documentation, among other non-scientific sources. This complements existing gaps on academic literature by providing "current" perspectives [17].

3.2 Research Questions to RR and GLR

Section 2 presents ORM-related problems from a real software system, which motivated us to conduct this research. Such problems could have been avoided if developers were aware of ORM code smells during development. So, to identify a catalog of smells, it is necessary to investigate what are the problems, their causes and potential solutions. Based on the aims of this research, we defined the following questions to guide both reviews (RR and GLR): **RQ1**: what are the problems and anti-patterns related to ORM in Java? **RQ2**: what are the causes for ORM-related problems in Java? **RQ3**: what are the suggested solutions to ORM-related problems in Java?

RQ1 aims to collect existing evidence of ORM-related problems, to group smells by problems, such as excessive data and N+1. Since smells are symptoms of problems, **RQ2** concerns with the root-causes of such problems, to identify smells for the catalog. And, **RQ3** aims to identify suggested solutions for the code smells.

3.3 Search Strategy

To perform the RR search, we used the *Scopus* search engine, as Cartaxo et al. [6] recommend. It aggregates research from different digital libraries to cover a large number of studies. Tests were performed with different search strings until a set was found that resulted in studies relevant to the `RQs`. We used the following search string over titles, abstracts, and keywords:

("ORM" OR "hibernate" OR "JPA" OR "eclipseLink" OR "openJPA") AND (problem* OR *smell OR anti-pattern* OR "performance"OR "maintainability") AND ("software develop*" OR "software engineering" OR "software project" OR "developer")

The GLR search was performed using Google, limiting the result set based on its ranking algorithm, as Garousi et al. [17] recommend. Some of the wildcards used in the RR search string were adapted. After tests, the search string was modified so that Google's search algorithm could find the most relevant results according to the RQs:

```
(ORM | hibernate | JPA | eclipseLink | openJPA) AND
( smell | problem | "anti-pattern" | performance |
maintainability)
```

3.4 Selection Procedure

We adhered to the following inclusion criteria to select sources, both in the RR and the GLR: **(1)** Sources that are directly ORM-related, in the software engineering context; **(2)** Sources with ORM problems or anti-patterns that are undetected during compile-time; **(3)** Sources that provide answers to at least one of the RQs; **(4)** Sources from 2010 onwards. Criterion **(2)** contains a restriction to select only evidence whose solutions are candidates for code smells. Moreover, criterion **(4)** avoids results prior to the JPA version 2.1, released in Dec/2009. Figure 1 shows the overall selection process and number of sources for both the RR and the GLR.

The RR search yielded 75 unique sources. Sources whose title was clearly unrelated or which did not meet the inclusion criteria were removed, so 24 sources remained. After reading and analyzing the abstracts, we reduced the list to 14 sources. When analyzing the entire studies content, we ended up selecting five sources.

Example 1 *ORM-related code to get a list of full names from part of the name*

```java
@Entity @Table("Person")
class Person{
    @Id @Column(name = id_person)
    private Integer id;
    private String name;
    @ManyToOne(fetch = FetchType.EAGER)
    private Address address
    @ManyToOne(fetch = FetchType.EAGER)
    private Gender gender
}
```

Listing 1 Person class

```java
public static List<Person> findPersons(String p){
 String jpql= "FROM Person WHERE name LIKE :p");
 ...
}
public static void main(String[] args) {
 List<Person> persons= findPersons("Jh");
 for (Person p : persons){
   System.out.println("Name = " + p.getName());
 }
}
```

Listing 2 Main class

Generated SQL statement:

```sql
-- Main SQL
SELECT * FROM Person p WHERE d.name ilike "%Jh%";
-- Additional queries performed:
SELECT * FROM Address d LEFT OUTER JOIN City c WHERE  c.id_city = d.city AND d.id_address = :id;
....
SELECT * FROM Gender g WHERE g.id_gender = :id;
....
```

SQL really needed:

```sql
SELECT name FROM Person p WHERE d.name ilike "%Jh%";
```

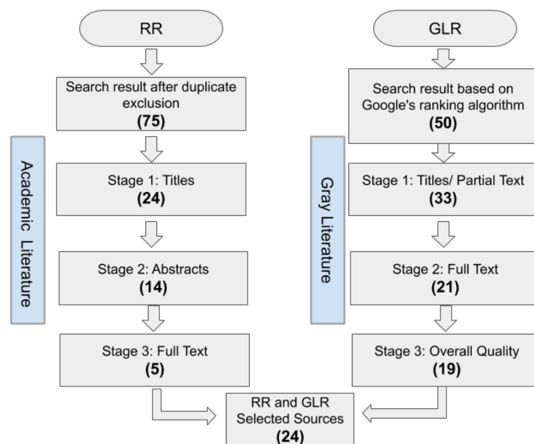


Figure 1: Selection procedure

The Google search for the GLR returned over 71 million records, so we defined limited effort [17] as a criterion for stopping the search. We used the first 50 relevant sources ranked by Google's algorithm. We first used the title and partial text available. The inclusion criteria was also verified and 33 sources were selected. Then, the full text was analyzed, resulting in 21 selected sources. To increase reliability, the evidence was assessed for overall quality, based in terms of clarity, detail, consistency, plausibility and alignment with the RQs, similar to previous works [46].

Sources that did not hold to minimum quality standards were excluded. For instance, sources without sufficient detail or with unclear explanations. Such standard could not be pre-determined as a criterion due to the diverse nature of the GLR, which requires case-by-case evaluation. At this stage, we excluded a source due to lack of plausibility [48]. It claimed to have discovered an undocumented way to improve JPA's performance, but some examples were already in the Hibernate documentation [40]. Another was excluded due to lack of detail [3]. A StackOverflow answer pointed to an existing source found by the GLR [35]. The full selection procedure is available online, with the list of all selected sources.[1]

3.5 Synthesis Procedure

Although the inclusion criteria only considers studies performed since 2010, the earliest study is from 2013 [49]. Only in 2016 we had more than one source in our search [9, 10]. Four sources have the same first author [7–10]. Furthermore, there is even a mention to a lack of studies regarding solutions to ORM problems, which is the focus of our work [10]. Regarding ORM problems, excessive data is the most cited problem in the RR [7–10]. The second one is $N + 1$, discussed in three studies [7–9].

In the GLR, 19 sources were selected, three of which are ORM framework documentation [33, 40, 44]; 14 are blog posts, eight of them [21–24, 30, 34, 35, 39] related to authors of ORM framework documentation; one Java developer forum post [1]; and one presentation by an author of the Hibernate documentation [31]. The most cited problem is $N + 1$ with 18 sources, followed by excessive data, with 15 sources. Some problems are explicitly mentioned in the sources [8, 28, 31], while others were inferred from the context [12, 33, 49]. Since code smells are symptoms of such problems, we infer their names from the different causes mentioned. We have summarized our results in an Evidence Briefing,[2] which is a one-page document that reports the main findings of a research.

[1] https://bit.ly/3d0O76v

[2] Available at https://bit.ly/31iEKfF.

4 A CATALOG OF ORM CODE SMELLS

In this section, we present the catalog of ORM-related code smells in Java. The definition of what a code smell represents is a subjective process, based on human experience and intuition [15]. For a practice to be considered a ORM-specific code smell, it should indicate a bad implementation choice, and suggest symptoms that may be indicative of something wrong in the code, thus indicating the need for refactoring. Therefore, based on the results of the reviews (RR and GLR) discussed in section 3, we extracted bad practices related to ORM in Java. We focus on the Java language in this catalog, since it has a standard API for ORM (JPA), that allows the uniform categorization of code smells from different ORM frameworks.

We defined a smell for each issue that had at least three mentions in different sources, resulting in eight code smells. We used this criterion to establish recurrence, as in the Rule of Three for defining patterns.[3] To delimit the scope of the research, we focused on the most recurring problems to explore them in detail, and also propose solutions. In this paper, we only present four smells, focusing on the most frequently mentioned issues, for brevity. The remaining are summarized at the end of this section. The full catalog is available online.[4] We present each smell with a short description, followed by a potential fix in the form of a refactoring, according to the best practices found in the reviews. Finally, we discuss and provide further details. Table 1 presents all smells in our catalog, classified by their problem type and followed by the evidences from both RR and GLR from which we extracted it.

4.1 Eager as a Fetch Strategy In Class-level (Static) Relationships

4.1.1 Smell Description. Using Eager fetching in attributes representing class-level relationships means that the related object is always retrieved from the database, even when never used. This might cause performance and maintainability issues and it prevents overriding the fetching strategy dynamically at the query level. Example 2 shows a code snippet in which the `Person` entity is unnecessarily retrieved when getting the student enrollment, due to the use of Eager as a fetch strategy. We use `...` to denote that code may exist before and after the snippets we show.

Example 2 Eager Smell

```
1 @Entity
2 class Student{
3     @ManyToOne(fetch = FetchType.EAGER)
4     private Person person;...
5 }
6 public static void main(String[] args) {
7     ...Student d = findStudentById(1);
8     d.getEnrollment();...
9 }
```

Generated SQL statement:

```
SELECT * FROM Student d LEFT OUTER JOIN Person p
ON     p.id_person = d.id_person WHERE d.id=1;
```

[3]http://wiki.c2.com/?RuleOfThree

[4]https://bit.ly/3aPOcu5

4.1.2 Refactoring Suggestion. Developers should use Lazy instead of Eager, to avoid always loading all the related objects. This way, they can retrieve data on demand, through additional queries. When needed, the object can be loaded in advance (Eager) at the query level through the Join Fetch clause or at the class level, by the `@NamedEntityGraph` annotation introduced in JPA 2.1.[5] Example 2 shows how to fix this, together with the resulting SQL query. The fix basically changes the fetching strategy to Lazy, thus avoiding unnecessary Joins with the `Person` entity.

Example 3 Fix Eager Smell

```
1 @Entity
2 class Student{
3     @ManyToOne(fetch = FetchType.LAZY)
4     private Person person;...
5 }
6 public static void main(String[] args) {
7     ...Student d = findStudentById(1);
8     d.getEnrollment();...
9 }
```

Generated SQL statement:

```
SELECT * FROM Student d WHERE d.id=1;
```

4.1.3 Discussion. According to the Hibernate documentation [40], using the Eager fetching strategy in a static way (defined at the class-level) is often a bad choice. This smell relates to the following risks: (i) Data of the related object is always loaded, even if not used. This causes unnecessary processing in the database. It can be worsened if the loaded object also has other Eager objects, which transitively generates unnecessary joins and queries [8]; (ii) If Join Fetch strategy is not used for all Eager relationships, a sub-query is performed, which may result in the $N+1$ problem [40]; (iii) An Eager relationship definition in the class-level cannot be overwritten to Lazy at the query-level. The opposite is possible, by performing a Join Fetch at the query-level [33].

Chen et al. [8] reports that changing the fetching strategy from Eager to Lazy for a class attribute with 10 records results in a 71% performance increase. Mihalcea [35], who is one of the authors of the Hibernate documentation, reports that using Eager in a static way is a *code smell*. Most of the time it is used to simplify development, but long-term performance issues are not considered.

Despite the fact that the reference documentation of both Hibernate version 5 [40] and Eclipselink [44] do not recommend its use, currently Eager is the default fetching strategy for `ManyToOne` and `OneToOne` mappings in Java ORM frameworks. According to Mihalcea et al. [40], this is due to the JPA specification. Before the JPA, Hibernate used Lazy by default for all of its mappings. When JPA 1.0 was released, it was believed that not all providers would use proxies to implement JPA. This could cause errors when lazily retrieving information in a closed connection, throwing the `LazyInitializationException`. Since ORM frameworks implement JPA, they follow the specification, but recommend against using Eager for `ManyToOne` and `OneToOne` mappings. Other mappings are Lazy by default. `ManyToOne` and `OneToOne` mappings

[5]https://docs.jboss.org/hibernate/jpa/2.1/api/

Table 1: ORM Code Smells catalog. RR lists evidence from the Rapid Review, while GLR relates to the Grey Literature Review

Num	Problem Type	ORM Code Smell	RR	GLR
1.	Excessive Data	EAGER AS A FETCH STRATEGY IN CLASS-LEVEL (STATIC) RELATIONSHIPS	[7–9]	[1, 12, 18, 20, 24, 28–31, 34, 35, 39–41, 44]
2.	Excessive Data	DATA RETRIEVAL WITHOUT PROJECTION FOR READ-ONLY	[9]	[18, 24, 31, 35]
3.	Excessive Data	UNNECESSARY UPDATING OF THE ENTIRE ENTITY	[7, 9, 10]	
4.	Excessive Data	NOT USING ORM PAGINATION TO AVOID UNNECESSARY RESULT TRANSFER		[18, 29, 31, 33, 35]
5.	$N + 1$	LACK OF JOIN FETCH IN ORM QUERIES TO RETRIEVE OBJECTS WITH EAGER ATTRIBUTES	[8, 9]	[1, 12, 18, 20, 22, 28–30, 35, 39, 41, 44]
6.	$N + 1$	ONE-BY-ONE: FETCH TYPE LAZY IN LOOPS	[7, 8]	[1, 12, 18, 20, 23, 28, 29, 33, 41]
7.	$N + 1$	UNILATERAL @ONETOMANY WITH INAPPROPRIATE USE OF COLLECTIONS	[49]	[30, 40]
8.	Others	NOT USING READ-ONLY QUERIES		[33, 35, 40]

using EAGER might lead to performance problems. For instance, when the retrieved object contains large data (binaries or images) or also has other EAGER relationships [9]. This smell might also cause maintenance issues. It can be easily fixed early during development by changing to LAZY. However, if we consider an already deployed system, in production, if other use cases use that class, expecting the object to be fully loaded, changing it to LAZY might result in throwing the `LazyInitializationException`. To avoid this, it is necessary to check all methods using the changed class, which relates to the Shotgun Surgery smell [15], where a small change in the code requires changes in several other classes.

The excessive data problem also increases when EAGER is used with `OneToMany` and `ManyToMany` associations, which represent collections and by default are LAZY. Using this strategy, the query may perform a *Cartesian Product* by limiting the speed of the query to the number of associated records, or causing the $N + 1$ problem, performing N additional queries [32].

Therefore, we consider the use of EAGER as a fetch strategy in class-level (static) relationships as a code smell. We can conclude that this is a bad implementation choice and may cause future issues related to performance and maintainability. Therefore, any class attribute representing an association between entities using `@ManyToMany`, `@OneToMany`, `@ManytoOne`, or `@OneToOne`, annotated with `FetchType.EAGER` is an instance of this smell. Moreover, for MANYTOONE and ONETOMANY relationships, we also consider as a smell those that are not explicitly annotated with `FetchType.LAZY`. MANYTOMANY and ONETOMANY with EAGER associations are worse, because querying an object results in retrieving N other objects in the relationship.

4.2 Not Using ORM Pagination to Avoid Unnecessary Result Transfer

4.2.1 Smell Description. Retrieving all records from a database table might result in excessive data and performance issues, specially when records are not fully used or necessary. Example 4 shows a code snippet with an example, in which all records from the `Student` entity from the year 2020 are retrieved, but only ten records are effectively used by the view layer, assuming that the `students` method is called with such value.

Example 4 Paged Smell

```
1 public List<Student> findStudents(int year){
2  ...hql = "FROM Student d WHERE year = :year";
3  Query q = entityManager.createQuery(hql);...
4 }
5 public List<Student> students(int year, int page,
      int limit) {
6  int fromIndex = (page - 1) * limit;
7  List<Student> students = findStudents(year);
8  return students.subList(fromIndex, Math.min(
      fromIndex + limit, students.size()));
9 }
```

Generated SQL statement:

```
SELECT * FROM Student d WHERE year=2020;
```

4.2.2 Refactoring Suggestion. The JPA specification provides the `setFirstResult(n)` and `setMaxResults(n)` methods. Both provide pagination features that can be used when performing an ORM query. Example 5 shows how to retrieve only ten `Student` records requested by the view layer, to avoid performance problems.

4.2.3 Discussion. To obtain better results in the ORM-generated SQL statements, it is important to also consider the number of records being retrieved, besides the number of columns, queries and joins [35]. We can use pagination to specify a limit on the number of retrieved records by a query [33]. For instance, we might use methods such as `setFirstResult(int)` to specify the offset, while `setMaxResults(int)` allows to specify the limit [18].

In the view layer, information is usually presented to the user in a paginated form. This visualization is directly linked to how we configure ORM data pagination [29]. For instance, for an entity with 1,000,000 records, we might only display a subset of information on a web page, such as 100 records per page. Therefore, to avoid loading all records, we can use `setFirstResult(1)` and `setMaxResults (100)` to retrieve only the first 100. Another issue related to not

Example 5 Fix Paged Smell

```
1  public List<Student> findStudents(
2   int year, int fromIndex, int limit) {
3   ...hql = "FROM Student d WHERE year = :year";
4   Query q = entityManager.createQuery(hql);
5   q.setFirstResult(fromIndex);
6   q.setMaxResults(limit);...
7  }
8  public List<Student> students(int year, int page,
       int limit) {
9   int fromIndex = (page - 1) * limit;
10  return findStudents(year,fromIndex,limit);
11 }
```

Generated SQL statement:

```
SELECT * FROM Student d WHERE year=2020
LIMIT  10 OFFSET 10;
```

using pagination is that the amount of data tends to grow over time [35], which might lead to performance issues.

This way, we consider an ORM code smell to not use pagination in data collections when the recovered records are not fully used. This indicates potential future problems related to performance according to the database growth.

4.3 Lack of Join Fetch in ORM Queries to Retrieve Objects With Eager Attributes

4.3.1 Smell Description. ORM query languages, such as HQL and JPQL, allow using objects with Eager relationships without performing Join Fetch clauses for the related objects. This causes the $N + 1$ problem, in which the framework performs one query to retrieve the object, and N additional queries for the related Eager objects, until all data is retrieved. Example 6 shows how a query to return a list might result in N additional queries.

Example 6 *Join Fetch Smell*

```
1  @Entity
2  class Student{
3   @ManyToOne(fetch = FetchType.EAGER)
4   private Person person;...
5  }
6  public List<Student> findStudents(int year){
7   ...hql = "FROM Student d WHERE d.year= :year";...
8  }
9  public static void main(String[] args) {
10  List<Student> d = findStudents(2020);...
11 }
```

Generated SQL statement:

```
SELECT * FROM Student d WHERE d.year=2020;
-- Additional queries performed:
SELECT * FROM Person p where p.id_person = 1;
SELECT * FROM Person p where p.id_person = 2;
....
```

4.3.2 Refactoring Suggestion. A potential fix is to change the fetch type from Eager to Lazy. When this is not possible, due to maintenance issues regarding dependencies, one can use Join Fetch to retrieve all information in a single query. Example 7 shows how to use Join Fetch to retrieve the Person entity trough join.

Example 7 Fix *Join Fetch Smell*

```
1  @Entity
2  class Student{
3   @ManyToOne(fetch = FetchType.EAGER)
4   private Person person;...
5  }
6  public List<Student> findStudents(int year){
7   ...hql.append("FROM Student d ");
8   hql.append("JOIN FETCH d.person p ");
9   hql.append("WHERE d.year = :year");...
10 }
11 public static void main(String[] args) {
12  List<Student> d = findStudents(2020);...
13 }
```

Generated SQL statement:

```
SELECT * FROM Student d INNER JOIN Person p
ON     p.id_person = d.id_person WHERE d.year=2020;
```

4.3.3 Discussion. We observe that the ORM framework might need to perform additional queries to retrieve information and properly instantiate the object. This can lead to performance problems, depending on the number of performed queries to complete the operation [37]. We previously discussed that using Eager in class-level relationship is a code smell. Therefore, the recommendation would still be to avoid using Eager as a fetching strategy at the class-level. When this is not possible, due to maintenance problems regarding dependencies, developers can use Join Fetch for all Eager attributes to avoid the $N+1$ problem. The maintenance problem occurs when the existing code expects that the related object is already loaded. This way, changing to Lazy can result in problems such as the `LazyInitializationException` from Hibernate [40]. The bad way to handle this exception is to use the Open Session in View pattern or enable the `hibernate.enable_lazy_load_no_trans` option [38]. These are considered anti-patterns, because they only treats the symptoms and does not solve the real cause of the problem [36]. The best way is to fetch all necessary associations before closing the persistence context through the Join Fetch clause [40]. This is particularly important to avoid the $N + 1$ problem [35].

Regarding the fix in Example 7, a developer analysis is important, because the number of joins can also negatively affect performance [4]. There are differences between performing a common Join and a Join Fetch. Fetch is JPA-specific [24]. It tells the persistence provider to also initialize the association on the retrieved object besides performing the join between the two entities.

Based on the above, we consider a code smell in ORM query languages the lack of Join Fetch when using entities with the Eager fetching strategy, as it might cause the $N + 1$ problem.

4.4 One-by-one: Fetch Type Lazy in Loops

4.4.1 Smell Description. When one needs to retrieve an attribute from an object inside a loop, if its fetch strategy is Lazy, additional queries are performed for each loop iteration. Example 8 demonstrates a query for retrieving a list of `Person` and subsequently obtaining the `students` attribute inside a loop. This generates extra queries to load the Lazy relationship for each loop iteration.

4.4.2 Refactoring Suggestion. One solution to this problem would be to preload the information using Join Fetch to retrieve all data in a single query. This consists of adding a Join Fetch statement, similarly as Example 7. Another solution is to use the `@BatchSize(n)` annotation so that the ORM framework uses the SQL IN operator, grouping and decreasing the number of additional queries. Example 9 shows a solution using `@BatchSize`.

Example 8 One-by-one Smell

```
1  @Entity
2  class Person{
3   @OneToMany(fetch = FetchType.LAZY)
4   private List <Student> students;...
5  }
6  public static void main(String[] args) {
7   List<Person> persons = findAll();
8   for (Person person : persons){
9     assertNotNull(person.getStudents());
10  }
11 }
```

Generated SQL statement:

```
SELECT * FROM Person p;
-- Additional queries performed:
SELECT * FROM Student d where d.id_person = 1
SELECT * FROM Student d where d.id_person = 2
....
```

Example 9 Fix: One-by-one Smell with @BatchSize

```
1  @Entity
2  class Person{
3   @BatchSize(size=4) @OneToMany(fetch=FetchType.LAZY)
4   private List <Student> students;...
5  }
6  public static void main(String[] args) {
7   List<Person> persons = findAll();
8   for (Person person : persons){
9     assertNotNull(person.getStudents());
10  }
11 }
```

Generated SQL statement:

```
SELECT * FROM Person p;
-- Additional queries performed:
SELECT * FROM Student d where d.id_person in (1,2,3,4)
SELECT * FROM Student d where d.id_person in (5,6,7,8)
....
```

4.4.3 Discussion. The Lazy fetching strategy, used by default in the `@OneToMany` and `@ManyToMany` annotations in the JPA specification [35] retrieves data on demand. This means that the persistence context will not load the relationship entity until performing an operation on it [20]. However, when the code requests the data, usually the transaction and database session have already ended. Since there is no possibility of joining, the framework performs N other queries to retrieve the data [40]. Inside a loop, this might generate an excess number of queries and decrease system performance.

There may also be maintenance issues when solving this smell. One widely used strategy, which can be found in Stack Overflow, is to change the fetching strategy to Eager [28]. However, this only changes the problem, instead of fixing it. As mentioned in Section 4.1, using Eager at the class level might lead to excessive data issues. A better strategy is to follow the suggested fix in Section 4.4.2.

Therefore, we consider a smell the retrieval of an attribute using the Lazy fetching strategy from an object in a collection inside a loop. The following conditions apply: the code must reside inside a loop, which accesses an attribute annotated with Lazy; it must not be loaded in advance using Join Fetch; it does not contain the `@BatchSize` annotation in the relationship attribute.

4.5 Other Code Smells

As mentioned, we do not detail the entire catalog for brevity. Here we briefly present the remaining smells. **Data Retrieval Without Projection for Read-only**: Using ORM queries without using projection for read-only (when no persistence context management is required) can lead to performance issues, especially when retrieving large data, such as BLOB columns, or using unnecessary joins, causing incompatibility between required and recovered data. **Unnecessary Updating of the Entire Entity**: Updating all columns of an entity instead of only the modified ones results in excessive data updates. This might lead to performance issues, in particular when the table has non-clustered indexes, large binary data or a large number of columns. **Unilateral `@OneToMany` with Inappropriate Use of Collections**: Using the `@OneToMany` annotation without the corresponding `@ManyToOne` in the other side of the relationship, combined with inappropriate use of collections, might lead to the $N + 1$ issue. When a change in an element of a collection results in all related records being removed and added again, depending on the semantics used. **Not Using Read-Only Queries**: Retrieving objects from the database for read-only purposes, without informing this to the ORM framework. Since the object does not need to be managed by the persistence context, failing to use read-only queries causes unnecessary memory allocation when saving state and also makes fetching and flushing less efficient.

5 EVALUATING THE CATALOG

To evaluate the proposed catalog, we conducted a survey with developers that are familiar with Java ORM frameworks. We believe it is important to understand practitioners' opinions, as they might influence on how to address the topic, and how to present empirical evidence Devanbu et al. [11]. We understand that the survey captures developers' beliefs, which also impacts practice. In what follows we detail the survey methodology and results.

5.1 Survey Methodology

We applied a mixed strategy to announce the survey and collect responses. We used *snowballing* [19] to share the survey link with professionals who met the requirements and encouraging them to indicate other potential participants. Moreover, we also shared the survey on social networks, such as *Twitter* and *Reddit*. We also mined Stackoverflow to contact potential participants from those who interacted through questions and answers regarding ORM frameworks in Java. We used the Stack Exchange Data Explorer[6] to extract data from users who interacted with questions with the following tags: *ORM*, *Hibernate*, *EclipseLink*, *OpenJPA*, and *JPA*. After processing the data to extract the emails, we obtained a list of 368 emails from candidates to answer the survey.

We produced Portuguese and English versions of the survey. The online questionnaire is composed of: (i) an introduction explaining the requirements for answering the survey; (ii) a demographics section asking participants about their academic background, Java, and ORM experience; (iii) an introduction to ORM code smells; a set of 8 ORM code smells and their solution, with code examples; a field for general comments about the survey.

For each code smell, we used a 5-level Likert scale [43] to assess the agreement with problems and solutions. Besides, an optional text field allowed the participant to comment about the smell or the solution. No problems were reported to answer the survey.

5.2 Results

We received 86 responses, in which 48 come from the English version and 38 from the Portuguese. Most participants have a masters degree (41) followed by bachelor degree (35), high school diploma (5), vocational training (4) and doctorate degree (1). Most also have 5-10 years of experience in Java development (34) and 3-5 years (22) followed by more than 10 (18) and 0-2 (12) . Regarding their previous experience using ORM to develop Java applications, there was greater diversification among participants, with 3-5 (24) more than 10 (23), 0-2 (22) and 5-10 (17) developed applications.

Figure 2: Average agreement level

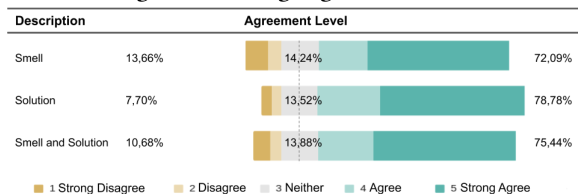


Figure 2 shows the average agreement level among all smells and solutions. The three percentages shown in the chart indicate the proportion of disagreements, neutral responses, and agreements respectively. In addition, dark and light green indicate the proportion of "Strongly agree" and "Agree" responses, while dark and light brown refer to "Strongly disagree" and "Disagree". The majority of respondents are positive (agree or strongly agree) about the smells (72.09%) and proposed solutions (78.78%). We noticed that agreement increases with the respondent experience level with ORM in Java. Thus, we also analyze our data separating the responses into **Experienced** (34) and **Beginner/intermediate** (52) developers. An Experienced developer is someone with more than five years of experience with Java, and with an extensive experience with ORM in Java (five or more developed applications). The remaining responses are considered as **Beginner/Intermediate** developers.

Figure 3: Average agreement by experienced developers

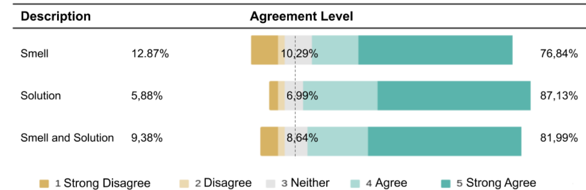


Figure 3 shows the average agreement level for experienced developers only. We notice an increase of 8.7% in the agreement level (strongly agree and agree) for both smells and solution combined (81.99%), when compared to the average of all participants. The agreement level for smells (76.84%) and solutions (87.13%) was also higher than the overall average. Regarding the less experienced developers (total of 52), there was actually a 5.7% decrease in the agreement level. The respondents agreed 68.99% with the smells and 73.32% for the solution, as shown in Figure 4.

Figure 4: Average agreement by beginner/intermediate developers

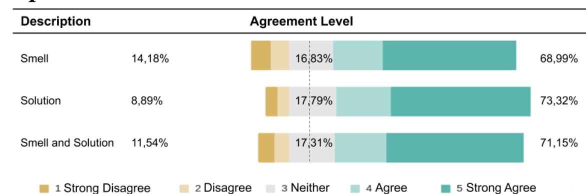
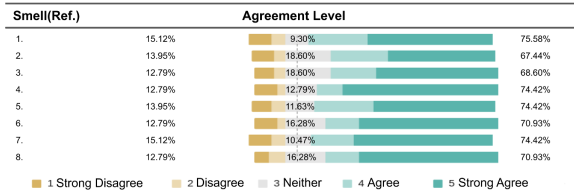


Figure 5 shows the agreement of each smell represented by numbers, related to the Table 1. The highest values refer to the 1. Eager as a Fetch Strategy In Class-level(Static) Relationships, with 75.58%, followed by 4. Not Using ORM Pagination to Avoid Unnecessary Result Transfer, 5. Lack of Join Fetch in ORM Queries to Retrieve Objects With Eager Attributes and 7. Unilateral @OneToMany with Inappropriate Use of Collections all with 74.42%. Regarding the solutions, Figure 6 shows that the highest agreement is with 4. Not Using ORM Pagination to Avoid Unnecessary Result Transfer, with 86.05%, followed by 1. Eager as a Fetch Strategy In Class-level(Static) Relationships and 5. Lack of Join Fetch in ORM Queries to Retrieve Objects With Eager Attributes both with 80,23%.

Comparing the averages between experienced (81.99%) and beginning/intermediate developers (71.15%), we observe that there was a positive variation of 15.23% for the experienced developers group. Experienced developers tend to work with the most complex part of the code, as well as problems of different causes [47]. We believe that the highest agreement level is due to the fact that

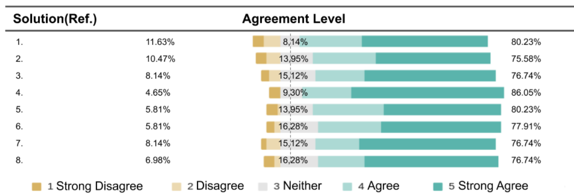
[6]https://data.stackexchange.com

Figure 5: Agreement level for each smell



experienced developers check these smells from ORM code and the problems generated for practical applications more often than beginner/intermediate developers.

Figure 6: Agreement level for each solution



We received some general comments about the catalog presented in the survey. Some participants showed appreciation for the catalog, for instance: "*Awesome! Everyone has to follow this. Highly recommend the solutions.*"; "*...It was enlightening and useful to learn such Bad Smells. My code will be better from now on.*", among others. Other participants expressed contempt for ORM, for example: "*Assumption that ORM is valid (& can be fixed up) is false. ORM needs to be eliminated...*"; "*SQL is better than ORM*".

6 THREATS TO VALIDITY

In this section, we present the threats to validity for our research.

Since RR is a lightweight technique, some of its procedures might present threats compared to a full-blown systematic literature review [5]. The GLR also has its threats, particularly due to its sources being non-scientific. Threats and limitations that apply to both RR and GLR consist on the selection procedure, which was conducted by a single researcher, and might cause selection bias; moreover, there was also no quality evaluation in the sources, which can reduce confidence in the evidences. Only one search engine was used for the RR. We mitigate this by using an engine that aggregates several databases. For GLR, limiting the results to the Google ranking algorithm imposes a hard limit on the number of analyzed resources, which might have discarded useful resources. Due to the practical nature of the research, we used the first 50 results, as it is often the case that users tend to focus on the first page, mostly.

Even though Java implements JPA, ORM frameworks might have particular issues, that are not captured by the catalog. This happens because we intend to have a general code smells catalog. Not every code smell needs refactoring, necessarily. For example, the use of Eager fetching in class-level may never lead to a performance or maintainability problem in cases where the information is always necessary in the relationship and database size is small. However, this code smell can often lead to problems, so a manual review by the developer is recommended before refactoring.

Although we state upfront the requirements for participating in the survey, we cannot guarantee that participants fulfill them. We also did not include the discussion of each smell, making it difficult for the participant to better understand the exceptions to the smell.

7 RELATED WORK

Previous studies propose approaches to catalog or detect smells in different contexts. Garcia et al. [16] discuss architectural smells, describing four representative smells encountered through reverse-engineering. Aniche et al. [2] provide a catalog of Model-View-Controller architectural smells defined by surveying 53 MVC developers. Fard and Mesbah [14] propose a set of 13 JavaScript code smells collected from various developer resources with metric-based approach to detect smells in code. Similar to those, our study has a domain-specific focus, that is, ORM-related code smells.

Some studies propose approaches to identify ORM problems. Nazario et al. [42] developed a framework that detects and reports 12 ORM problems identified through an observational study. The addressed problems related to compile- or runtime errors when integrating ORM code. In our study, we focus on problems that might lead to performance or maintainability issues. Chen et al. [8] propose a framework to detect ORM performance anti-patterns based on static code analysis. They later provide an automated approach to locate redundant data problems in ORM code [9], finding out that 87% of the transactions in their evaluation performed with redundant data. Węgrzynowicz [49] discuss five performance anti-patterns related to the usage of one-to-many associations in Hibernate. Most prior studies on ORM focus on performance anti-patterns. Nonetheless, code smells are not limited to such issues. To the best of our knowledge, our work is the first to present ORM code smells in a structured way, discussing their root causes.

8 CONCLUSION

ORM frameworks provide a conceptual abstraction by mapping database records to objects aiming to make developers focus on business logic. However, this abstraction may lead to several problems due to less-than-ideal implementation choices in ORM code. These symptoms that indicate low quality code are called code smells. In this paper, we propose a catalog of ORM code smells for Java in a systematic way through evidence found in the context of literature and practice. The catalog was evaluated regarding the agreement level of the smells and solutions presented to Java developers through a survey, obtaining 86 responses with an average agreement of 75.44%. There was an increase of 8.7% in the general agreement level with the catalog when we only consider responses from experienced developers. This might relate to the fact that they may have seen the problems generated by the reported smells in the catalog more often. We believe this catalog can be useful not only to researchers but also to practitioners. Presenting smells to Java developers might avoid future maintenance and performance problems. Future work can use the catalog to develop tools to automatically detect smells. We also intend to study ORM smells in frameworks for other programming languages.

ACKNOWLEDGMENTS

Leopoldo Teixeira was partially supported by CNPq (409335/2016-9) and FACEPE (APQ-0570-1.03/14), as well as INES 2.0,[7] FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

REFERENCES

[1] Sayem Ahmed. 2018. JPA Tips: Avoiding the N + 1 select problem. Retrieved Feb. 16, 2020 from https://www.javacodegeeks.com/2018/04/jpa-tips-avoiding-the-n-1-select-problem.html
[2] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. 2018. Code smells for model-view-controller architectures. *Empirical Software Engineering* (2018).
[3] Mat B. 2011. Hibernate performance - Stack Overflow. Retrieved Feb. 16, 2020 from https://stackoverflow.com/questions/5155718/hibernate-performance
[4] Christian Bauer, Gavin King, and Gary Gregory. 2016. *Java Persistence with Hibernate*. Manning Publications Co.
[5] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2018. The Role of Rapid Reviews in Supporting Decision-Making in Software Engineering Practice. In *EASE'18*.
[6] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2020. *Contemporary Empirical Methods in Software Engineering* (1 ed.). Springer, Chapter Rapid Reviews in Software Engineering.
[7] T. Chen. 2015. Improving the quality of large-scale database-centric software systems by analyzing database access code. In *ICDEW'15*.
[8] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. (2014).
[9] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 2016. 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* (Dec 2016).
[10] T. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. 2016. An Empirical Study on the Practice of Maintaining Object-Relational Mapping Code in Java Systems. In *MSR'16*.
[11] Prem Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief & Evidence in Empirical Software Engineering. In *ICSE '16*.
[12] Hippolyte Durix. 2020. Boost the performance of your Spring Data JPA application. Retrieved Feb. 16, 2020 from https://blog.ippon.tech/boost-the-performance-of-your-spring-data-jpa-application
[13] EclipseLink. 2019. What is Object-Relational Mapping. Retrieved Oct. 22, 2019 from https://wiki.eclipse.org/EclipseLink/FAQ/JPA#What_is_Object-Relational_Mapping
[14] A. M. Fard and A. Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *SCAM'13*.
[15] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley, Reading, MA.
[16] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. 2009. Identifying Architectural Bad Smells. In *CSMR'09*.
[17] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* (2019).
[18] Gustavo Gomes. 2016. Performance Improvement in Java Applications: ORM / JPA. Retrieved Feb. 16, 2020 from https://dzone.com/articles/performance-improvement-in-java-applications-orm-j
[19] Leo Goodman. 1961. Snowball Sampling. *Ann Math Stat* (1961).
[20] Chris Hut. 2015. Avoiding JPA Performance Pitfalls. Retrieved Feb. 16, 2020 from https://www.veracode.com/blog/secure-development/avoiding-jpa-performance-pitfalls
[21] Thorben Janssen. 2015. How to Improve JPA Performance | Rebel. Retrieved Feb. 16, 2020 from https://www.jrebel.com/blog/how-to-improve-jpa-performance
[22] Thorben Janssen. 2017. Solve Hibernate Performance Issues in Development. Retrieved Feb. 16, 2020 from https://stackify.com/find-hibernate-performance-issues
[23] Thorben Janssen. 2018. 3 Common Hibernate Performance Issues in Your Log. Retrieved Feb. 16, 2020 from https://www.baeldung.com/hibernate-common-performance-problems-in-logs
[24] Thorben Janssen. 2020. 7 Tips to boost your Hibernate performance - Thoughts on Java. Retrieved Feb. 16, 2020 from https://thoughts-on-java.org/tips-to-boost-your-hibernate-performance
[25] Rod Johnson. 2005. J2EE Development Frameworks. *Computer* (Jan. 2005).
[26] Josh Juneau. 2018. The Query API and JPQL. In *Java EE 8 Recipes*. Springer.
[27] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An exploratory study of the impact of code smells on software change-proneness. In *WCRE'09*. IEEE.
[28] Michal Marciniec. 2019. JPA: N+1 SELECT problem. Retrieved Feb. 16, 2020 from https://codete.com/blog/jpa-n-plus-1-select-problem
[29] Igor Medeiros. 2015. Java Persistence API: Otimizando a performance das aplicações. Retrieved Feb. 16, 2020 from https://www.devmedia.com.br/java-persistence-api-otimizando-a-performance-das-aplicacoes/32091
[30] Vlad Mihalcea. 2016. Hibernate Performance Tuning and Best Practices. Retrieved Feb. 16, 2020 from https://in.relation.to/2016/09/28/performance-tuning-and-best-practices
[31] Vlad Mihalcea. 2016. High-Performance Hibernate Devoxx France. Retrieved Feb. 16, 2020 from https://www.slideshare.net/VladMihalcea/high-performance-hibernate-devoxx-france
[32] V. Mihalcea. 2016. *High-Performance Java Persistence*. Vlad Mihalcea.
[33] Vlad Mihalcea. 2017. Performance Features. Retrieved Feb. 02, 2020 from https://www.eclipse.org/eclipselink/documentation/2.7/solutions/performance001.htm
[34] Vlad Mihalcea. 2019. EAGER fetching is a code smell when using JPA and Hibernate. Retrieved Feb. 16, 2020 from https://vladmihalcea.com/eager-fetching-is-a-code-smell
[35] Vlad Mihalcea. 2019. Hibernate performance tuning tips. Retrieved Feb. 03, 2020 from https://vladmihalcea.com/hibernate-performance-tuning-tips
[36] Vlad Mihalcea. 2019. The hibernate.enable_lazy_load_no_trans Anti-Pattern. Retrieved Jan. 19, 2020 from https://vladmihalcea.com/the-hibernate-enable_lazy_load_no_trans-anti-pattern
[37] Vlad Mihalcea. 2019. How to detect the Hibernate N+1 query problem during testing. Retrieved Mar. 22, 2020 from https://vladmihalcea.com/how-to-detect-the-n-plus-one-query-problem-during-testing
[38] Vlad Mihalcea. 2020. The best way to handle the LazyInitializationException. Retrieved Mar. 20, 2020 from https://vladmihalcea.com/the-best-way-to-handle-the-lazyinitializationexception
[39] Vlad Mihalcea. 2020. prevent JPA and Hibernate performance issues. Retrieved Feb. 16, 2020 from https://vladmihalcea.com/jpa-hibernate-performance-issues
[40] Vlad Mihalcea et al. 2018. Hibernate ORM 5.4.3.Final User Guide. Retrieved Feb. 16, 2020 from https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html
[41] João Munhoz. 2019. Hibernate and the n+1 selections problem - QuintoAndar Tech. Retrieved Feb. 16, 2020 from https://medium.com/quintoandar-tech-blog/hibernate-and-the-n-1-selections-problem-c497710fa3fe
[42] Marcos Felipe Carvalho Nazario, Eduardo Guerra, Rodrigo Bonifacio, and Gustavo Pinto. 2019. Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report. In *ESEM'19*.
[43] A.N. Oppenheim. 2000. *Questionnaire Design, Interviewing and Attitude Measurement*. Bloomsbury Academic.
[44] Oracle. 2015. Oracle TopLink JPA Performance Tuning. Retrieved Feb. 16, 2020 from https://docs.oracle.com/middleware/1212/core/ASPER/toplink.htm
[45] Guy Paré, Marie-Claude Trudel, Mirou Jaana, and Spyros Kitsiou. 2015. Synthesizing information systems knowledge: A typology of literature reviews. *Information & Management* 52, 2 (2015).
[46] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* (2013).
[47] T. Tsunoda, H. Washizaki, Y. Fukazawa, S. Inoue, Y. Hanai, and M. Kanazawa. 2017. Evaluating the work of experienced and inexperienced developers considering work difficulty in sotware development. In *SNPD'17*.
[48] Fabrício U. 2015. I discovered an undocumented way to improve JPA performance. Retrieved Feb. 16, 2020 from https://bewire.be/blog/i-discovered-an-undocumented-way-to-improve-jpa-performance
[49] P. Węgrzynowicz. 2013. Performance antipatterns of one to many association in hibernate. In *FedCSIS'13*.
[50] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *ICSE'13*. IEEE Press.

[7] http://www.ines.org.br