Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report

Marcos Felipe Carvalho Nazário
Federal University of Pará
Evandro Chagas Institute
Belém, Brazil
marcosnazario@iec.gov.br

Eduardo Guerra
National Institute for Space Research
São José dos Campos, Brazil
eduardo.guerra@inpe.br

Rodrigo Bonifácio
University of Brasília
Brasilia, Brazil
rbonifacio@unb.br

Gustavo Pinto
Federal University of Pará
Belém, Brazil
gpinto@ufpa.br

Abstract*—*Background: **Object-Relational Mapping (ORM) frameworks are regarded as key tools in the software engineer arsenal. However, developers often face ORM problems, and the solution to these problems are not always clear. To mitigate these problems, we created a framework that detects and reports a family of ORM problems.** ***Aims:*** **The aim of this work is to assess how practitioners perceive our framework, the problems, they face, and the eventual points for improvements.** ***Method:*** **We first report an observational study in which we curated 12 ORM-related problems, which are implemented in our framework. We then conducted a developer experience (DX) study with 13 developers (10 well-experienced and 3 students) to assess their experience with our framework to implement six ORM-related tasks.** ***Results:*** **All participants agreed that our framework helped them to finish the programming tasks. The participants perceived that our framework eases the ORM modeling, has precise error messages, and employs ORM best practices. As a shortcoming, however, one participant mentioned that some custom annotations are not very intuitive.** ***Conclusions:*** **Our findings indicate that developers are willing to use frameworks that catch ORM problems, which create opportunities for new research and tools.**

I. Introduction

Nowadays, developers often take advantage of Object-Relation Mapping (ORM) frameworks to provide a conceptual abstraction between objects in object-oriented languages and data stored in the underlying database [2]. ORM technologies intercede between object-oriented architecture system and the relational environment [8]. This constitutes, in effect, a "virtual object database" that can be handled from within the programming language.

The use of ORM frameworks greatly reduces the effort of not only communicating with a database but also dealing with basic database operations (e.g., insert, update, read, and delete), since changes to objects are automatically propagated to the corresponding database records. It then may come as no surprise that ORM frameworks are widespread in the software development industry. As an example, a survey[1] pointed out that 67.5% of Java developers use ORM frameworks to communicate with the databases. However, these ORM frameworks may be underused or overused. For instance, in a recent work, the authors observed that redundant data problems in ORM code could deteriorate the performance of an enterprise system [2]. Due to the lack of works in this direction, we believe that other concerns might exist.

Through our own experience in dealing with an enterprise system, we perceived that it is not uncommon for developers to misuse ORM code by, for instance, using the wrong annotation, or referring to the wrong column. We call such kind of problems as "ORM problems". Unfortunately, the development team did not have access to the production database (or replication of it) in the early stages of software development. Therefore, such ORM problems only became visible when the development team had to integrate the ORM code with the production database, which often happens during the deployment of the source code to the production environment. Based on our experience, we noted that ORM-related problems are very domain specific; therefore tools such as Findbugs [6] provide little or no guidance in this context.

After years dealing with ORM problems, we curated a catalog of these problems and automated their detection in a framework. This framework is underuse in our company for over five years now, and our experience suggests that several mapping problems have been avoided. Our framework consists of static analyzing ORM code looking for the ORM problems. We use bytecode instrumentation to (1) obtain the ORM code and to (2) create several assertions that check whether the ORM code contains one (or more) of the curated ORM problems. We also provide enhanced error messages that aid developers to fix the ORM problem found. After cataloging the mapping problems and proving a framework to automatically identify and report these problems, we conduct a developer experience (DX) study [4] with 13 participants to better understand the framework's limitations and eventual points for improvements.

II. Background

Java Annotations. Annotations are one of the most used Java programming language construct [3]. Introduced in version 5, annotations enable metadata inclusion directly into the source code. This characteristic permits maintaining the source code and metadata together, improving application maintenance. Annotations can be seen as a "stamp" that can, e.g., help compilers to detect other classes of errors. Several Java frameworks take advantage of annotations. Particularly relevant to this study is the JPA framework.

Java Persistence API (JPA). Released in 2009, JPA is the specification for object-relational mapping with Java. ORM

[1]https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/

978-1-7281-2968-6/19/$31.00

frameworks such as Hibernate (that implements the JPA specification) are growing in demand in developers communities [8]. Hibernate alleviates developers from implementing routines demanded to address the relationship between objects and database relations. JPA (and consequently Hibernate) makes extensive use of annotations. Consider Listing 1, which presents an entity class, i.e., a class that represents a table in a relational database that does not have any other responsibility.

Listing 1. Example of a JPA Entity class

```
1  // Imports omitted
2  @Entity
3  @Table(schema = "public", name = "COLOR")
4  public class Color {
5
6      @Id
7      @Column(name = "CODE", nullable = false)
8      private String code;
9
10     @Column(name = "NAME", length = 80,
           nullable = false)
11     private String name;
12
13     @Column(name = "ENABLE", nullable = false)
14     private Boolean enable;
15 }
```

In this example, there are several annotations including: (1) `@Id` to map a primary key, (2) `@Column` to map a column, and (3) `@Table` to map a table. An extensive list of annotations (and their purposes) is available in the JPA official website[2].

III. The Context

In 2010, the first author of this work was the software architect in a development team composed by one software coordinator, one data analyst, and seven software developers. The majority of them had at least three years of software development experience. The enterprise system under development automated the process for obtaining credit lines for the first and second sectors of the economy.

The defined architecture was chosen considering the expertise of the development team and the technologies approved by the company. The enterprise system was written in Java, following a typical MVC architecture. In particular, the enterprise system used Hibernate for dealing with ORM code, and JSF, a web framework. As of today, the enterprise system has 306 classes with 33,457 lines of code. There are 82 entity classes in the enterprise system, totalizing 15,731 lines of code (47.02% of the total of lines of code). The enterprise system has only 10 testing classes created by the development team. The other testing classes are created by the framework, using the strategy described in Section V-B. Our framework, on the other hand, has 51 classes, comprehending 6,971 lines of code. The software development process followed two main steps:

Step 1: Creating the schema. After the requirement was stable it was sent to the database administrator, which was in charge of creating the database schema (in a DDL format). The database administrator also specified restrictions such as the null or not null, data type, data length, as well as the primary keys and foreign keys. It is important to note that the client required the development team to follow a prefix pattern in every column. For instance, a column of type date should be prefixed with "DT_" and a string should be prefixed with "TX_". The database administrator modeled the schema in his local workspace. Since the database administrator worked on several requirements at the same time, his workspace was always ahead of and more complex than the workspace of the development team. The database administrator was also responsible for ensuring that the different environments (e.g., testing and production) were seamlessly replicated. Since the database administrator was often working in more than one schema, it usually took some time to replicate them.

Step 2: Creating the entity classes. After the database schema was done, the requirements along with the schema were sent to the development team to implement the entity classes. At this moment, the development team had no access to the database of the database administrator. Then, the development team had to rely on the text-based schema and the requirements to manually create the entity classes. The object-relational mapping was done using the annotations provided by JPA.

We also used the Bean Validation framework[3], which provides additional annotations that aid developers to pose constraints in the entity classes (e.g., the `@NotNull` and `@NotEmpty` annotations). Afterward, the development team had to create classes that communicate with the database. At this moment, the developers had to test the entity classes using different strategies, such as mocking the database or using an in-memory database. Once the communication with the database was established, the development team was able to create tests cases to make sure that the access the database was working properly. After that, several ORM problems popped up frequently. The next section presents an analysis performed in the recurrent problems to categorize and classify them.

IV. The Mapping Problems

During our experience implementing the entity classes and database integration, we often faced some ORM problems. We organized them in terms of: 1) Problems with the database schema and 2) Problems with the entity classes.

A. Problems with the database schema

In this category, most of the problems were due to the fact that the database administrator had to implement the schema manually. They are described next.

- **P1** Employing the wrong prefix in the column's name (e.g., prefixing with a TX_ when an EN_ was needed);
- **P2** Misspelling the names of schemas, tables, or columns (e.g., names with accents);
- **P3** Forgetting to create constraints (e.g., did not create a foreign key constraint);
- **P4** Forgetting to apply the same DDL at different environments (e.g., testing and production).

B. Problems with the entity classes

Since the development team had only access to the DDL of the database, they could not rely on reverse engineering tools to create the data entity classes. Instead, they have to

[2]https://jcp.org/en/jsr/detail?id=317

[3]https://beanvalidation.org/1.0/spec/

re-implement the DDL in Java code. During this process, a new class of problems emerged. They are listed below.

- **P5** Misspelling the name of the schema, tables, and columns (similar to **P2**, but in the Java code);
- **P6** Mismatching JPA annotations (e.g., using `@Column` when a `@JoinColumn` was needed);
- **P7** Employing the wrong constraint attributes in a `@Column` or a `@JoinColumn` annotations (e.g., not defining the `nullable` or `length` attributes);
- **P8** Forgetting to use an important annotation (e.g., a `@Temporal` annotation in a date datatype, a `@Enumared` in an Enum datatype, or a `@Table` in a class). This leads an implicit default mapping which is not always the one desired;
- **P9** Forgetting to define at least one primary key using the `@Id` or the `@EmbeddedId` annotation.
- **P10** Forgetting to implement the `Serializable` interface. According to the JPA specification, all entities must implement the `Serializable` interface;
- **P11** Forgetting to use Bean Validation annotations to reinforce JPA annotations (e.g., not using Bean Validation to constraints such as `nullable`, `length` and others from `@Column` and `@JoinColumn`);
- **P12** Naming the attributes or methods of the class in non-compliance with the Java standard, e.g., the use of characters with an accent. This happens because developers usually copy the name of the attributes directly from the requirements and forgot to remove them);

It is important to note that, by default, JPA supports explicit and implicit mapping. The explicit mapping is when the developer declares the JPA annotations in the source code on classes and fields. The implicit mapping is when the developer does not declare the annotations in the code then JPA uses conventions for classes and fields mapping. When developers forget to declare an annotation, the usage of implicit mapping can lead to problems such as **P7**, **P8**, and **P12**.

C. The consequences of these problems

During the integration with the database, some of the (so far hidden) problems reported in the previous sections started to appear to the development team. These problems eventually lead to the following *consequences*:

- **C1** When using `@Column` instead of `@JoinColumn` (i.e., **P7**), the JPA tries to serialize the whole foreign key object, and save it in a cell. Depending on the version of the JPA provider, this could lead to an error called data truncation[4] or an error of not allowed property[5]. When the mapping is done correctly, the JPA only stores the value of the foreign key in the column.
- **C2** When using implicit table mapping (i.e., **P8**), the JPA per default assumes that the default schema is "public", leading to an error called table not found[6].
- **C3** When using implicit column mapping for an Enum datatype (i.e., **P8**), the JPA assumes that the default behavior is to save the ordinal position declared at the Enum constant in the column. However, in general, when using an Enum type, developers want the `name` of the Enum constant as the value[7]. In summary, the column expects a text and a number is sent instead. The number is then converted to text and persisted in the database. This behavior does not raise any errors or warnings. Developers may only perceive the wrong behavior when inspecting the rows in the database.
- **C4** When using implicit column mapping for the date datatype (i.e., **P8**), JPA assumes that the default behavior is to save the only date without the clock time. This consequence do not raise any error or warning, but may not always produce the correct behavior, and only looking at the rows in the database one could see this problem.

These problems showed that the development team had serious issues that needed to be handled. Unfortunately, these issues were hardly prioritized because the company was more interested in delivering new features of the enterprise system to the clients. The development process was often behind the schedule, which generated several extra hours often used to fix the listed problems. However, no effort was placed to find the root cause of them. This decision has placed a lot of burden and stress in the development team.

V. The Proposed Solution

The aforementioned problems (and their consequences) share a common root cause: they were all driven by the mismatch between the database schema and the manual mapping done in the entity classes. In this work, we mitigate this problem by proposing a framework focused on finding and reporting ORM problems. Our framework tackles these problems by implementing three modules:

1) A custom classes and annotations module (§ V-A).
2) An automatic generation of unit tests module (§ V-B).
3) An improved error report module (§ V-C).

A. Custom classes and annotations module

Our framework provides custom classes and annotations. Some of the custom annotations are wrappers of other annotations, in particular, because most of the developers that joined the development team were unaware of the Bean Validation annotations available (which has more 20 native annotations). Our customization kept the same behavior of the native annotations, but start with the `@Verify` prefix. For instance, the `@Email` annotation was wrapped in the `@VerityEmail` custom annotation. This small change allows one to quickly auto-complete the kind of annotation needed. This approach was aiming to reduce errors like **P11**.

Although some custom annotations were implemented to work as a wrapper of the Bean Validation annotations, some other annotations were introduced, such as the `@VerifyDuplicityRecord` annotation. For instance, in our company, almost every requirement has a duplicity rule. It indicates that duplicate records are not allowed in the database. In general, two approaches are used to implement this requirement. In the first one, the developer had to query for all stored records in the table, which is a very computation-intensive

[4]https://stackoverflow.com/q/27756137
[5]https://stackoverflow.com/q/4121485
[6]https://stackoverflow.com/q/4365857
[7]https://stackoverflow.com/q/2751733

operation, in particular, because the ORM framework has to map each database record with the corresponding object. In the second approach, the developer has to write a tailored SQL query (with fewer columns) to verify the existence of duplicate elements on the database. Since this second approach is less costly than the first one (it does not need to map rows into objects, and does not select all columns), we automated this behavior in the `@VerifyDuplicityRecord` annotation.

Finally, our framework also provides custom classes that help developers by providing default behavior. For instance, we provide the `PersistenceObject` class, which every entity class should inherit from. This class offers a default implementation for the `equals()`, `hashCode()`, `toString()`, `clone()`, and `compareTo()` methods, which mitigate misuses such as wrong hashing algorithm or wrong sorting criteria. It is also straightforward to change the behavior of these methods. If a new field is added or removed in the entity class, the methods are `equals()` and `hashCode()` are automatically updated accordingly.

B. Automatic generation of unit test modules

In this module, our framework automatically generates unit test classes. It uses the Java Reflection API to achieve this goal. When generating unit test classes for entity classes, our framework checks whether the entity classes, for instance, (1) implement the `Serializable` interface, (2) have the default Java methods (e.g. `equals()`, `hashCode()`), or (3) employ other custom annotations.

Moreover, the framework also provides a testing superclass named `TestModelBase` that is responsible asserts the behavior of the subclasses. Any test class should then inherit from the `TestModelBase` class to guarantee a default way for testing the entity classes. Although every entity class should have an equivalent test class that extends from the `TestModelBase`, we observed quite often that developers forgot to create the test class. We then created an assertion that checks if the testing class exists; if that is not the case, the framework generates one automatically. More concretely, the entity `Color` should have a test class named `TestColor` which is a subclass of `TestModelBase`; if the testing class does not exist, then the framework creates one during its execution. After the framework is done with its analysis, all automated generated unit classes are discarded. We opted to discard the tests because (1) they are very cheap to create (around three seconds in our enterprise system) and (2) they do not have any other role in the project.

This superclass employs the *groups of test* strategy, i.e., the tests are categorized into groups and invoked in precedence order. In essence, the `TestModelBase` class checks if all the assertions of each group holds true. If one group of tests fail, its dependencies are not invoked. The six groups of tests (with their precedence order) are the followings: (1) the *serializable* group, (2) the *default methods* group, (3) the *bean validation* group, (4) the *connection* group, (5) the *annotation* group, and (6) the *JPA* group. Due to space constrains, we provide at most three assertions per group of group.

The serializable group (3 assertions). This is responsible to check if the field `serialVersionUID` is present since all JPA entities must be serialized (Fix **P10**). More precisely:

1) If the field is declared with `final` and `static` modifiers.
2) If the field has `private` visibility.
3) If the field has an unique number among the other entity classes.

The default methods group (6 assertions). This group is responsible to assert the behavior of JPA entities, for instance:

1) If the entity class is a subclass of `PersistenceObject`.
2) If at least one instance field of the entity class is annotated with `@ApplyComparable`.
3) If the name of the entity class has accents (Fix **P12**).

The annotation group (4 assertions). This group is responsible for checking all the remaining custom annotations, for instance:

1) If the class is annotated with `@VerifyRelationships`.
2) If any instance fields are annotated with `@VerifyDuplicityRecord` (Fixes **C5**)
3) If each entry of `@VerifyRelationship` is up to date. It verifies if each entry references a valid relationship or if a new one must be added (Fixes **C6**).

The JPA group (24 assertions). This group is responsible to check misuses in the JPA annotations, for instance:

1) If an instance field annotated with `@Embedded` is also annotated with `@Valid` in order to verify cascading (Fixes **P11**).
2) If an instance field annotated with `@JoinColumn` is also annotated with `@ManyToOne` or `@OneToOne` (Fixes **P6**).
3) If an instance field annotated with `@Column` or `@JoinColumn` has the `nullable` attribute set to `false` is also annotated with `@VerifyNotNull` (Fixes **P11**).

There are also other assertions that do not pertain to any particular group, such as verifying if there is an open connection with the database. With these test strategies, our framework eliminates most of the commons ORM problems that were already mentioned before. Overall, for a small JPA entity class with two attributes and 12 annotations, our framework creates one unit test class with 60 assertions. If any of these assertions fail, the project is not able to compile. Therefore, developers are required to follow the guidelines proposed in the framework.

C. Enhanced error messages module

Throughout our experience using the JPA framework, we observed that the error messages are not always intuitive. Take as an example of an error related to data truncation, already mentioned as a consequence **C1** of **P7**. The traditional error message raised is the following:

```
Data truncation: Data too long for column
'movie' at row 1 org.hibernate.exception.
DataException: could not insert:
[com.model.Timetable]
```

This error message, although provides a glimpse of the problem (e.g., the data inserted on column 'movie'), places

little guidance on how to fix it. Moreover, this error message misses an opportunity to exploit the ORM problem that is, indeed, the root cause of the problem. Therefore, developers may not easily perceive its fix. To mitigate non-intuitive error messages like the above one, in our work we place a particular effort in creating errors messages that pinpoint the causes and also suggest the fix. If our automatic generated unit tests flag any ORM problems, our framework produces a customized error summary by adding descriptive messages. As an example, the previous default JPA error message is replaced for the following customized error message.

```
The attribute 'movie' in the class 'com.
model.Timetable' is annotated with @Column
but should be annotated with @JoinColumn.
```

We believe that our set of assertions, as well as the curated error messages, can be useful to help developers overcome object-relational mapping problems. Moreover, since most of the assertions are created automatically, we also alleviate the developers with a non-trivial effort. Taking into consideration these three modules, our framework has 51 classes, comprehending 6,971 lines of code.

VI. The Developer Experience (DX) Study

Our framework has been applied in the enterprise system described in Section III for five years now. After the framework became integrated into our development process, the number of ORM problems found in production reduced significantly. The successful usage of the proposed framework in our enterprise system is initial evidence that it can bring benefits to projects that present the same characteristics.

To exploit the benefits (and eventual limitations) of our framework, we conducted a Developer Experience (DX) study. According to Fagerholm et al [4], DX could be defined as an approach for capturing how developers think about their activities within their working environments. DX assumes that an improvement in the developer experience could have a positive impact on, e.g., sustained teams and projects performance.

The goal. The goal of this study is to verify if a developer (with very different skill levels) can use the proposed framework successfully, identifying its strong points and opportunities for improvement.

The participants. We recruited 13 participants to perform this experience, in which 10 work in the industry, and 3 undergrad students. The participants were recruited following a convenience sample: they are close contacts to the authors of this paper, although none of them had previous experience with our framework. Two of the participants, however, had no experience with ORM code. On average, they have 6.7 years of experience with Java programming (4.9 years with ORM programming). We refer to them as P1–P13.

The experiment. We asked the participants to perform six ORM tasks, varying from creating columns, creating a 1 to N relationship, to creating an N to N relationship. The participant had to create other entity classes throughout the experiment. The experiment was conducted locally with 5 participants, and remotely with the other 8 participants. For the remote participants, we provide them a virtual machine with our framework configured in the Eclipse IDE. We also provide a basic Java project, which has one single entity already implemented, for guidance purposes. On average, the participants took four hours to complete the tasks (min: 2h, max: 8h). All participants completed the tasks, and all tasks were completed correctly. After the tasks, we asked the participants to share their code. We also interview them to better understand their perceptions regarding our framework. Two interviews were conducted in person, and the remaining ones by online tools (e.g., WhatsApp and Skype). The audio was recorded and later transcribed.

TABLE I
Description of the participants

#	Job role	Java Exp.	ORM Exp.	Time taken
P1	Software Engineer	3 years	1.5 year	3.5 h
P2	Student	1.5 year	7 months	3.5 h
P3	Software Engineer	14 years	14 years	2.0 h
P4	Software Engineer	16 years	14 years	2.5 h
P5	Student	6 months	None	4.0 h
P6	Student	4 years	None	2.5 h
P7	Help Desk	3 years	1.5 year	8.0 h
P8	Software Engineer	10 years	6 years	6.0 h
P9	Help Desk	3 years	1.5 year	6.0 h
P10	Help Desk	4 years	1.5 years	5.5 h
P11	Software Engineer	7 years	7 year	2.0 h
P12	Software Engineer	12 years	8 year	3.5 h
P13	Software Engineer	9 years	9 year	2.0 h
	Average	6.7 years	4.9 years	3.9 h
	Standard Deviation	5.0 years	5.0 years	1.9 h

Replication Package. We provide the materials used in this study (e.g., the virtual machine, the transcripts, etc), as well as the source code of our framework: https://tiny.cc/qtfeaz.

A. DX Findings

Benefits. In terms of benefits, the participants perceived that the framework **eases the ORM modeling**. For instance, P3 mentioned that "*Every day I do object-relational mapping, and this framework could help me to remember what is missing*". Moreover, the enhanced error messages were perceived as **precise**. P5 mentioned that "*After I did an initial study to understand what the framework was about, the error messages were useful to indicate how to implement the ORM tasks.*". Still, all participants agree that the enhanced error messages helped them to conclude the experiments. One particularly interesting finding is that two participants (P5 and P6) finished the tasks without any ORM experience. When we interviewed them, we perceived that they were able to accomplish the task by following the suggestions of the enhanced error messages. Similarly, although P8 has many years of experience, he is not working with JPA anymore. This participant took six hours to finish the tasks. He mentioned that "*I had to remember how to do the mapping in JPA since I do not work with it anymore*". This participant also highlighted the enhanced error messages, saying that "*some messages helped more than others, but in the end, everything went worked smoothly, even not remembering well the correct mapping*". Still, P11 mentioned that our framework can be useful for **training purposes** since the team becomes used to the standards enforced by the framework. Finally, P4 highlighted that the framework follows the ORM **best practices**.

Challenges. There are also some challenges in using our framework. According to P4, some **custom annotations were not intuitive**. For instance, P4 mentioned that "*It took me a little longer to understand that the* `@VerifyRelationship` *annotation must be declared inside the* `@VerifyRelationships` *annotation.*". When declaring the `@VerifyRelationship` annotation outside the `@VerifyRelationships` annotation, this participant received a non-intuitive error message, suggesting that a `@VerifyRelationship` annotation has to be added in the entity class, which made no sense since the annotation was already there. This happened because we assumed that our custom annotations were straightforward to use. Important to note that, for all the more than 60 assertions, the participants only faced problems with the one related to the `@VerifyRelationships` annotation. Moreover, eight participants mentioned that the **documentation is scarce**. P7 stressed that this was particularly the case of the documentation of the annotations. Nevertheless, it is important to note that, although the documentation was not sufficient in some points, all participants were able to complete all the tasks. Interestingly, we did not experience these two problems in our company (the non-intuitive annotation and the scarce documentation). This could be explained due to the fact that the architect of the framework as always available to answer eventual questions. Nevertheless, when considering the perspectives of making the framework useful to the general public, these problems become more relevant. Moreover, P4 also mentioned that the framework **poses many constraints**, such as the use of a prefix in the column's name (e.g., "TX_" for strings) or restricting the use of lazy for fetching strategies (i.e., a collection is fetched only when the application invokes an operation upon that collection).

Manual inspection. We manually inspected the source code that our participants made during the experiments. We perceived that their code is sightly uniform when it comes to the implementation of the entity classes. This finding highlights another benefit of our framework: even when not explicitly asked, our participants (with very different skill levels) were able to achieve the expected solution.

VII. Limitations

On the technical side, our framework is limited to Java projects that use version 1.5 or higher. Our framework is also limited to ORM problems with the JPA framework. Still, our framework requires access to the source code, therefore it could not help much if only the binaries are available.

In terms of the experiment, our study is limited by the number of participants (only 13 developers participated in our study). However, the participants have different levels of experience (which foster diversity) and all of them were well experienced with ORM frameworks (which makes their response more trustworthy). Moreover, during our experiment, we perceived that our framework is not only helpful for those that have already experience in ORM since two students with no previous ORM experience were able to finish the tasks.

Finally, one may argue that our framework might not be relevant today since some of the ORM problems reported were due to our experience working with the JPA, some years ago. However, when browsing Q&A websites today, we still see these problems occurring. For instance, the question stackoverflow.com/q/55466284 asked in April 2019 (two months ago as of this writing), could be fixed by our framework (see C1). Therefore, we believe that developers are still demanding tools that aid them to deal with ORM problems.

VIII. Related Work

There are many works that propose tools to aid developers to find and fix errors. jPET [1] is a white box test-case generator that performs reverse engineering of the test-cases generated at the bytecode level. Randoop [9] generates unit tests for Java code using feedback-directed random test generation. EvoSuite [5] is an engine that automatically creates test cases with assertions for Java classes. SpongeBugs is a Java tool that automatically fixes warnings raised by SonarQube [7]. Other works focused on problems related in ORM code [2]. To the best of our knowledge, there is no research work that introduces and assesses tools that aid developers to find and fix ORM problems.

IX. Conclusion

ORM problems are commonplace in the developer landscape. Unfortunately, many of the tools available to find and fix errors do not help much, since ORM problems are often domain specific. In this work, we share our experience by (1) categorizing 12 ORM problems, (2) introducing a framework that aid developers to identify these problems, and (3) conducting a developer experience study with 13 participants. We observed that our framework helped them to finish the coding tasks. The participants also perceived some benefits while using the framework, such as the precise error messages or the adherence to the ORM best practices.

Acknowledgments. We thank the participants for collaborating in this research and the reviewers for their helpful comments. This was is partially supported by CNPq (#406308/2016-0), FAPESP (#2014/16236-6) and PROPESP/UFPA.

References

[1] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gomez-Zamalloa, and S. Gutierrez. jpet: An automatic test-case generator for java. In *18th WCRE*, pages 441–442, Oct. 2011.

[2] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. 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 TSE.*, 42(12):1148–1161, 2016.

[3] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of java language features. In *36th ICSE*, pages 779–790, 2014.

[4] F. Fagerholm and J. Münch. Developer experience: Concept and definition. In *18th ICSSP*, pages 73–77, June 2012.

[5] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *6th ICST*, pages 362–369, Mar. 2013.

[6] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[7] D. Marcilio, C. A. Furia, R. Bonifcio, and G. Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *19th SCAM*, 2019.

[8] E. J. O'Neil. Object/relational mapping 2008: Hibernate and the entity data model (edm). In *SIGMOD*, pages 1351–1356, 2008.

[9] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *22nd OOPSLA*, pages 815–816. ACM, 2007.