HBSniff: A Static Analysis Tool for Java Hibernate Object-Relational Mapping Code Smell Detection

Zijie Huang, Zhiqing Shao*, Guisheng Fan*, Huiqun Yu, Kang Yang, Ziyi Zhou

Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China

Abstract

Code smells are symptoms of sub-optimal software design and implementation choices. Detection tools were actively developed for general code smell related to coupling and cohesion issues, but such tools cannot capture domain-specific problems. In this work, we fill the gap in data persistence and query code quality by proposing HBSniff, i.e., a static analysis tool for detecting 14 code smells as well as 4 mapping metrics in Java Hibernate Object-Relational Mapping (ORM) codes. HBSniff is tested, documented, and manually validated. It also generates readable and customizable reports for every project. Moreover, it is beneficial to Mining Software Repository (MSR) research requiring large-scale analysis since project compilation is not needed for detection.

Keywords: Code Smell, Object-Relational Mapping, Hibernate, Static Analysis, Object-Oriented Programming

1. Introduction

Code smells (*i.e.,* symptoms of sub-optimal design and implementation choices [1]) are related to determining factors of software quality such as change- and error-proneness [2]. Compared with software defects, code smells

*Corresponding Authors.

Email addresses: hzj@mail.ecust.edu.cn (Zijie Huang), zshao@ecust.edu.cn (Zhiqing Shao), gsfan@ecust.edu.cn (Guisheng Fan), yhq@ecust.edu.cn (Huiqun Yu), 15921709583@163.com (Kang Yang), zhouziyi@mail.ecust.edu.cn (Ziyi Zhou)

are more likely to be underestimated, and they have long life-cycles, which means they may impose negative effects in the long run [2]. Thus, code smell detectors were actively developed to capture severe issues that may hinder maintainability. Recently, researchers found that general smell detection tools were not able to capture more specific problems related to domain-specific code [3], and domain-specific smells (*e.g.,* for data persistence [4, 5, 6, 7]) were attracting more attention from researchers and practitioners.

To facilitate Object-Oriented Programming (OOP), practitioners use Object-Relational Mapping (ORM) frameworks which bridge database and application by filling the gap of data mapping and persistence [4, 8]. Despite their flexibility and capability, there exist various challenges in the application of ORM [8] including the metamorphic class and table inheritance [9, 10], the inconsistency in data structure [11], and the uncontrollable propagation of relational data retrieval [12]. Consequently, ORM usage is regarded as a double-edged sword [4, 8] or even an anti-pattern [13]. However, recent studies suggested ORM need not affect performance if used properly [4], and practitioners need more static analysis tool support to help them with development [3]. In response, related studies [13, 14] outlined several smells and refactoring strategies to cope with them. In most cases, they either did not provide tools, or the tools were early prototypes and requires project compiling, which is not ideal [15, 16] for large-scale analysis over real-world systems. Thus, we fill the gap by proposing a static analysis tool called HBSniff (HiBernate Sniffer) for ORM code smell detection. Similar to related studies [3, 12, 13, 14, 17, 18, 19], we use the trending Java Hibernate[1] framework as the context of our implementation.

The contributions of our work include:

- We implement a Hibernate code quality static analysis tool called HBSniff for detecting 14 code smells and calculating 4 mapping metrics.
- Compared with the state-of-the-art dynamic analysis tool [14], we implement more code smells, and our tool supports the detection of projects using Java version greater than 1.7.
- We manually validate our tool on 5 open-source projects and 1 commercial project, and we prove the high impact of 7 out of 8 performance smells in a case study.

The highlights of HBSniff are:

[1]https://hibernate.org/

- Compilation is not required for the evaluated project.
- Test cases and documentations for every smell detector and metric are included.
- The tool generates a customizable and readable report for every project.
- The code is open-sourced on GitHub.

The rest of this paper is organized as follows. In Section 2 we introduce the problems, background, and summarize related work. Section 3 presents the architecture and the implemented smells, while Section 4 outlines the evaluation results, the advantage of our tool, as well as further research opportunities. In Section 5 we present illustrative examples of our tool. Finally, Section 6 concludes the paper and describes future research.

2. Problems and Background

In this section, we introduce the background and related work of ORM code quality analysis as well as code smells.

2.1. Java ORM and Hibernate Query Language (HQL)

Listing 1: Example of An ORM Entity (User.java)

```java
package com.example.blog.models;
import javax.persistence.*; import java.util.*;
import java.io.*;           import lombok.Data;
@Data @Entity @Table("users")
public class User implements Serializable {
    @Id @Column
    private Integer id;
    @Column(unique = true)
    private String email;
    @Column
    private String password;
    @Column
    private String name;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "user")
    private List<Post> posts = new ArrayList<>();

    public User() {} // No-Arg Constructor
}
```

Java ORM frameworks implement the Java Persistence API (JPA[2]) in order to map the tables, columns, and relationships of trending RDBMS (Relational DataBase Management Systems, *e.g.,* MySQL[3]) to the OOP-driven classes, attributes, and inheritance [8]. Listing 1 shows an example of an ORM entity class. Classes annotated with `@Entity` indicates that it is an entity in the ORM context, while the `@Table` annotation specifies its corresponding RDBMS data table. Moreover, `@Id` could be used to specify the unique identifier (in most cases, the corresponding field of the Primary Key of RDBMS data table), and relational annotations like `@ManyToOne`, `@OneToMany` could be used to describe relationships between entities with `FetchType` (*e.g.,* `EAGER, LAZY`) specified to determine whether data should be fetched in advance or on demand.

Listing 2: Example of A User Query (Method Extracted from UserRepository.java)

```java

public List<User> getUserByEmail(String queryEmail){
    EntityManager em = connection.createEntityManager();
    String hql = "from User u where u.email=:email";

    Query query = em.createQuery(hql, User.class);
    query.setParameter("email", queryEmail);

    List<User> results = query.getResultslist();
    return results;
}
```

Listing 2 demonstrates an example of an HQL query using the entity class of Listing 1, which is a method to retrieve users by their email addresses. The HQL template including the `email` parameter is defined in the `String` variable called `hql`, and it is later populated by the method parameter called `queryEmail`. Finally, the populated HQL query is executed by Hibernate, and the results are returned in a `List` collection of the `User` entities. HQL is a SQL-like query language designed for ORM [13, 20]. HQL could either be generated by ORM or specified by developers. Then, ORM will translate HQL to executable SQLs for RDBMS. Later, the results of the queries will be processed by ORM and presented with the form of OOP in the context of

[2] https://www.jcp.org/en/jsr/detail?id=338

[3] https://www.mysql.com/

JVM (Java Virtual Machine). We focus on the human-written HQLs in this work.

2.2. ORM Code Quality Analysis and Opitimization

Since ORM relies heavily on automatic relational data retrieval and mapping, performance issues are primary concerns of practitioners. Procaccianti *et al.* [21] found that ORM approaches can introduce a 70% increase in execution time, and they significantly increased energy consumption. However, there existed conflicting opinions indicating ORM need not affect performance if used properly [4]. Thus, researchers intended to find best practices to ensure the appropriate usage of ORM. Chen *et al.* [22] proposed a cache optimizers for Hibernate-based web applications, which improved the throughput up to 138% comparing to default caching strategy. Singh *et al.* [23] exploited multi-objective genetic algorithms to improve ORM performance by optimizing ORM configurations. Lorenz *et al.* [9] compared the performance of 3 different mapping strategies and provided visualizations of radar charts as conclusions. Chen *et al.* [3] conducted an empirical study on ORM code changes, they found that ORM codes were frequently modified, and such modifications lacked static analysis tool support. Meurice *et al.* [11] proposed a detection approach to address inconsistencies in ORM code after database schema change. Nazário *et al.* [24] proposed a development framework to solve 12 problems with mapping problems as well as their consequences.

2.3. Domain-Specific Code Smells

The state-of-the-arts of code smell detection tools mainly focused on general code smells such as coupling, cohesion, and complexity issues [25], such as Feature Envy, Spaghetti Code, and Complex Class. However, since recent work revealed there may be "other fish in the sea [1, 26]" that may impose uncaptured impact to software maintainability, various domain-specific code smell detection tools were proposed. For example, in the domains related to our work, Aniche *et al.* outlined various smells of model, view, controller codes in web applications [27], and the quality of Structured Query Language (SQL) queries [5, 6, 7] were also discussed.

In terms of ORM smells that we implement, Holder *et al.* [10] proposed a metric suite to measure ORM mapping code complexity. Silva *et al.* [14] proposed a set of rules to check if Hibernate entity codes follow JPA specifications. Loli *et al.* [13] summarized ORM code smells proposed in prior studies [3, 12, 17, 18, 19] as well as in grey literature, and they investigated the agreement of developers towards the definition of smells. Results showed most developers agree with the definitions and severity.

3. Software Framework

3.1. Software Architecture

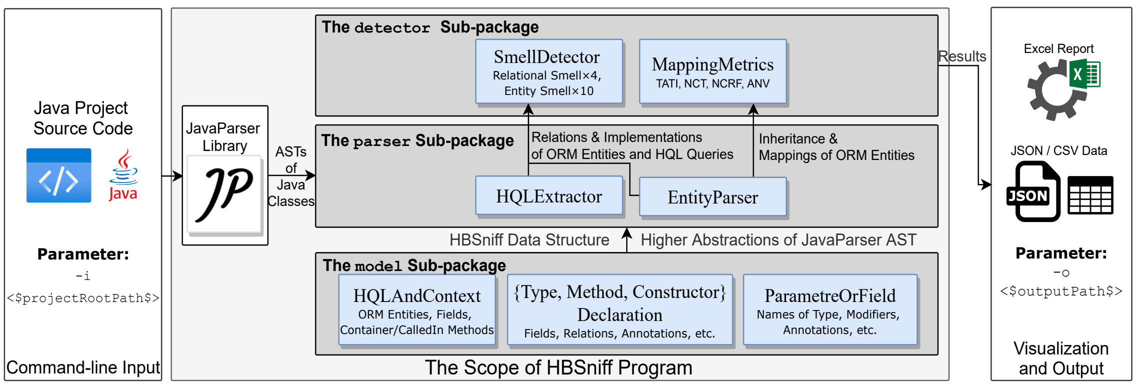


Figure 1: The general architecture of HBSniff.

HBSniff is a Java project using Maven[4] as dependency management and building tool which consists of 3 major modules (sub-packages), *i.e.*, `model`, `parser`, and `detector`. The general architecture is depicted in Figure 1. The light gray colored box represents the scope of HBSniff code, while the dark gray colored boxes are sub-packages of HBSniff. The blue-colored boxes are classes in the sub-packages, and the white-colored boxes are external libraries and resources.

First, users specify the path of the project to detect as input (the `-i` parameter) and the path of the output directory (the `-o` parameter). Then, the program constructs higher level abstractions (whose data models are available in the `model` sub-package) in the `parser` sub-package using JavaParser[5]. Meanwhile, it locates HQL in the context of the method and class containing the method call. Moreover, it finds the method that calls the method containing HQL (`CalledIn` methods in Fig. 1). Afterwards,

[4] https://maven.apache.org/
[5] https://javaparser.org/

the generated `models` will be used to populate the context of code smell `detectors`. Finally, the detection and metric evaluation will be performed, and the results will be converted to EXCEL reports as well as `csv` and `JSON` data. The demonstrations of the outputs are available in Section 5.

The implementation of the 10 smell detectors and 4 mapping metrics follows their definitions, which will be described in the next Subsection. Each detector class extends a class called `SmellDetector` which provides standard interface methods and basic capabilities (*e.g.,* loading all available HIBERNATE entities). Since we only have 4 closely related `MappingMetrics` to detect, they are now implemented in a single file. We will exploit a strategy similar to `SmellDetector` if more metrics are developed in the future.

Apart from the detectors, we also implement 2 `parsers` of HQL and HIBERNATE entities to avoid operating directly on the lower-level JAVAPARSER ASTs, which may be more challenging to comprehend and maintain. The `parsers` are designed to retrieve relations, inheritance, mappings, and other implementation details from the original JAVAPARSER ASTs. We introduce the 2 `parsers` in the next paragraphs.

The `EntityParser` consists of several methods aiming to parse Java classes to the JAVAPARSER `CompilationUnits` , and convert `CompilationUnit` to HBSNIFF-defined `TypeDeclaration` in the `model` package. Each `TypeDeclaration` refers to a Java class (including HIBERNATE entities), which contains its nested fields, its relations with nested types, its class- and method-wide annotations, its constructors, and so on. Meanwhile, HBSNIFF also defines the abstractions of methods and constructors.

The `HQLExtractor` locates the `createQuery` method call which indicates potential HQL usage [28], and generates `HQLAndContext` objects containing the HQL query `String` and its corresponding context, *e.g.,* the signature of the method containing HQL (*i.e.,* container method of `HQLAndContext` in Fig. 1), the available types and their fields (presented in the `ParametreOrField` objects from the `model` sub-package) of the entities in the FROM phrase of the HQL query, the methods which call the container of HQL (*i.e.,* `CalledIn` methods of `HQLAndContext` in Fig. 1), and so on.

3.2. Inter-Entity Relational Smells Detected

Relational smells summarized in [13] are inter-entity smells caused by inappropriate usage of data retrieval strategies in entity relationships. Some of the relational smells are related to the N+1 performance issue. The N+1 problem occurs when an application retrieves a parent entity from the database, and then loops through a collection field of the entity containing N other entities. HIBERNATE may generate a query for every iteration to retrieve smelly entities, which means we call to the database recurrently. In total, the application will call the database once for every row (*i.e.*, for N times) returned by the original query, and the plus one refers to the original query. This problem could lead to performance issue if the size of N is large. However, reducing N is not acceptable since we may need that large amount of data. The appropriate way to address it is to correctly configure the relationships between entities and the strategies of data fetching. For example, using the `LAZY FetchType` with specified `BatchSize` for on-demand batch retrieval. The relational smells are listed as follows.

(1) Eager Fetch [12, 17, 18]: HIBERNATE preloads all fields annotated with the `EAGER FetchType` when the data class is initialized, even if some of them will never be accessed. To avoid performance issues (*i.e.*, retrieving too much data in advance), data should be retrieved on demand.

(2) Lacking Join Fetch [12, 18]: Fields annotated with `EAGER FetchType` should be joined by `join fetch` in HQL to be retrieved through one query using join. Otherwise, such fields would be retrieved by N additional queries if the parent object is initialized, resulting in the N+1 problem.

(3) One-By-One [12, 17]: A collection annotated with `@OneToMany` or `@ManyToMany` using `LAZY FetchType` will be fetched one-by-one in every loop iteration. `@BatchSize` should be involved to load data on demand and in batch.

(4) Missing ManyToOne [19]: Using `@OneToMany` annotation in a field without `@ManyToOne` presented on the other side of the relationship may also lead to the N+1 problem.

3.3. Intra-Entity and Application Smells Detected

Entity smells are caused by inappropriate definition or application of entity fields and methods. **Smells (5)** to **(13)** are summarized in [14], while **Smell (14)** is described in [13]. The entity smells are listed as follows.

(5) Collection Field: Collection fields should use `Set` instead of `List` due to performance concern, *e.g.*, an insertion after deletion in a `List` may cause HIBERNATE to remove all the entities and re-insert them.

(6) Final Entity: Using `final` classes as entities would disable the proxy functionality of HIBERNATE to enhance the performance of lazy loading. Thus, the `LAZY FetchType` will fall back to `EAGER`, and no warning or error message will be thrown by HIBERNATE. As a result, the performance will be harmed silently.

(7) Missing No Argument Constructor: A no argument constructor should be implemented for HIBERNATE to generate an entity object using reflection, otherwise HIBERNATE will use Java reflection to initialize entities, which will consume more resources. Moreover, if HIBERNATE is used as a provider of JPA, it will throw an `org.hibernate.InstantiationException`, and the application may crash since it may not be able to handle it.

(8) Missing Identifier: Identifier field should be specified to uniquely determine an entity. Otherwise, comparators of objects may be confused when dealing with data objects containing identical data from different rows.

(9) Missing Equals Method: The default `equals` method compares the reference of objects, which is not ideal for comparing entities, especially for collection-related operations. The lack of an appropriate `equals` method will cause failure in reconnecting the detached entities, which may cause data persistence problems such as duplication.

(10) Missing HashCode Method: `HashCode` is vital for collections such as `HashSets` to determine equivalent entities. The consequence of this smell is similar to Missing Equals Method.

(11) Using Identifier in Equals or HashCode Methods: The identifier should not be used in `equals` and `hashCode` since all transient objects may be equal because their identifiers could be null. The consequence of this smell is similar to Missing Equals Method.

(12) Not Serializable: Entities which would leave the domain of JVM (*i.e.,* detached for data export) should implement the `Serializable` interface. Otherwise the serialization may fail, and Java will throw an `exception` called `java.io.NotSerializableException`.

(13) Missing Accessor Methods: Although HIBERNATE does not require accessor methods, JPA specification recommends implementing publicly visible getters and setters to access and update private fields.

(14) Local Pagination: Built-in pagination of ORM should be used to fetch the data of each page instead of fetching all data and locally split them for pagination.

3.4. Mapping Metrics Implemented

The 4 Mapping metrics [10] are designed to evaluate data redundancy and performance of entities related to inheritance. Thresholds to identify a smell should be investigated further.

Table Accesses for Type Identification (TATI): The number of tables needed to identify the requested type of entity. Higher TATI indicates more queries will be executed to construct an object of the entity.

Number of Corresponding Tables (NCT): The number of tables that contain data of an entity, which measures object retrieval performance. The impact of higher NCT is similar to higher TATI.

Number of Corresponding Relational Fields (NCRF): The number of relational fields in all tables that correspond to each non-inherited non-key field of an entity, which measures change propagation. Higher NCRF indicates more queries will be executed to change data of an entity since there exists data redundancy in terms of the relational fields.

Additional Null Values (ANV): The number of null values in the row of union superclasses, which measures the data redundancy. Higher ANV indicates more storage space is used to store null values since entities affected by such a problem are likely to be stored with other entities (with nonidentical fields) in the same table.

4. Implementation and Empirical Results

HBSniff could be executed as a command-line program under JDK (Java Development Kit) version 1.8 and above with a line of command, *e.g.*, `java -jar HBSniff-1.6.8.jar -i <projectRootPath> -o <outputPath>`. The tool is shipped together with unit tests for all smells and metrics implemented and documentations for both developers and users. However, since it is also a tool for practical usage, we test it on real-world projects.

4.1. Manual Validation of Detection Results

We perform smell detection over 5 open-source projects and 1 commercial project (`CP`). The brief introduction of the projects are listed in Table 1. Prior study [14] constructed a dataset of 77 projects for evaluation, however, the authors found most of them were toy and example projects. To generate a more practical dataset, we pick 3 non-toy projects from [3, 14] having actual purpose and functionality. We also randomly pick the 4th and 5th project by locating `createQuery` method calls using GitHub search to find projects performing potential HQL execution. The 6th project is used to confirm if our tool can be used in a more realistic scenario. The smells are manually validated by the 1st and the 5th author independently. The

Project	Purpose	Entities	I%	R%
WeixinMultiPlatform [14]	Content management system.	26	100.00	38.46
Jpa-issuetracker [14]	Development issue tracker.	6	100.00	16.67
Broadleaf Commerce [3]	E-commerce framework.	162	100.00	71.60
Devproof Portal [6]	Blogging platform.	22	100.00	72.73
2ndInvesta [7]	Invest management.	16	100.00	62.50
CP (Commercial)	Order processing.	27	100.00	77.78

Table 1: Projects analyzed. I% refers to entities affected by intra-entity smells. R% refers to entities affected by relational smells.

detection is all accurate and there is no missing case. The analyzed results show that HIBERNATE-based projects are heavily affected by ORM smells, which indicates the need of analyzing empirically the impact of these smells to the quality of software in large-scale.

However, we fail to find an appropriate project for assessing the 4 metrics. Nevertheless, we implement the HIBERNATE-based examples of the original paper [10], which is also available with database generation code (ddl) in the example folder in our source code. Note that since recent versions of HIBERNATE do not support mixed inheritance strategy[8], our implementation is slightly different from the original paper. Finally, the 4 metrics are verified by unit tests and manual evaluation of the sample project [9].

4.2. Impact of ORM Performance Smells

The smells we detect are either maintainability-related (*i.e.,* **smells (8)-(13)**) or performance-related. Since the evaluation of maintainability impact relies on empirical studies over large-scale dataset, it is not within the scope of our work. However, we are still interested in the significance of the implemented performance smells, which could be measured by benchmark. Thus, we measure the impact of the **smells (1)-(7)** and **(14)** by retrieving or editing relational data in different scales (*i.e.,* 50, 500, and 50,000 instances of relational entities).

We construct 3 entities (*e.g.,* A, B, C) for every smell. Entity A is the parent, while B is a child of A, and C is a child of B. The parent entities

[6]https://github.com/devproof/portal
[7]https://github.com/2ndStack/2ndInvesta
[8]Mixing inheritance is not allowed. https://hibernate.atlassian.net/browse/HHH-7181
[9]https://tinyurl.com/2as46jx8

Smell	10 × 5 Instances			100 × 5 Instances			10000 × 5 Instances		
	Clean	Smelly	Impact	Clean	Smelly	Impact	Clean	Smelly	Impact
(1)	202	242	0.20	209	1457	5.97	304	130462	428.15
(2)	134	247	0.84	161	1469	8.12	692	123464	177.42
(3)	120	243	1.03	275	1474	4.36	2027	123190	59.77
(4)	122	240	0.97	268	1461	4.45	2013	124036	60.62
(5)	131	271	1.07	255	1561	5.12	507	140353	275.83
(6)	207	507	1.45	211	2743	12.00	344	139279	403.88
(7)	248	250	0.01	1484	1496	0.01	123465	125355	0.02
(14)	194	269	0.39	350	590	0.69	583	2791	3.79

Table 2: The performance of smelly and clean entities. The unit of the performance in the Clean and Smell columns is millisecond.

own the relations. Except for the foreign key field (`parent_id`), the entities have only 1 additional MySQL `VARCHAR(255)` column called `name` mapped as a corresponding `String` field in Java. Every instance of Entity B owns 5 instances of entity C as children, and we alter the number of the children of entity A (*i.e.,* instances of entity B) in {10, 100, 10,000} to measure the impact of smells to software systems with different data scale. We present a 3-level relation because we intend to recover a more practical scenario by using multi-level nested relation.

To ensure a comparable and fair performance score, each test is performed 10 times, and we re-execute the program rather than performing the queries in loops since Hibernate built-in cache cannot be disabled, and it may underestimate the impact. Afterwards, we present the medians of the performance data in Table 2. The clean column refers to the entity which is not affected by the smell concerned, while the smelly column refers to the entities affected by the smells. The impact column measures the differences between the two performance divided by the performance of clean entities.

The hardware environment of this experiment is consistent, *i.e.*, AMD Ryzen 7 4800H CPU, DDR4 16GB 2666MHz RAM, and 512GB SSD. We use MySQL as the database. All queries are performed on Hibernate 5.4.31 and Java 11. To clarify, we do not test the performance impact of the combinations of smells. For example, in the evaluation of the **pagination smell (14)**, we implement `LAZY FETCH` with `BatchSize` specified (and thus the entity is not affected by the prior smells), and we retrieve the first page since it is more likely to be the most visited page. However, **smell (7) Miss-**

ing No Argument Constructor is an exception since we intend to find out the impact of reflection-based constructor generation in more entities, and thus we test this smell on `EAGER Fetch` entities.

From the impact column, we can easily conclude that performance smell could impose more impact if more relational entities present in queries. Meanwhile, except for **smell (7) Missing No Argument Constructor**, almost all smells cause significant decline in performance if there are more than 500 instances of relational entities. Moreover, even in the case of 50 instances of instances of relational entities, the decline of performance caused by the smells may also reach 20% to 145%. Thus, we believe such smells are worth refactoring.

4.3. Comparison with Related Tools

Difference with Respect to [14]. Prior study [14] proposed a design rule checker which is capable of detecting 9 out of 10 intra-entity smells (except for Local Pagination). However, the checker requires the analyzed project to compile, and its upstream parser (DesignWizard [10]) provide full support only for the class files compiled with JDK version less than or equal to 1.7. JDK 1.7 is no longer supported[11] by its manufacturer since April 2015. Compared with DesignWizard, JavaParser is a static analysis based Java AST parser that updates weekly and supports the language features up to the latest Java 15. Moreover, we compile 2 projects in the datasets in Table 1 to verify the results, and we fix some issues in its implementation, *e.g.,* we can detect the cases of using identifier annotations in accessor methods, calls of parent methods by `super` in `equals` and `hashCode`, and we do not treat missing `equals` and `hashCode` as an occurrence of **smell (11)** since default methods compare object references instead of attributes, and so on.

The 3 Unimplemented Data Usage Smells Mentioned in Section 4.5 of [13]. The original source of the 3 smells [18] used static analysis to locate method calls of queries, and analyzed the accessed data using dynamic analysis. We do not implement them since static analysis is not able to profile execution. However, we will extend our work to find trials of redundant usage, *e.g.,* locating `findAll` and `update` operations of fetching a whole entity or table. To achieve this goal, a large-scale empirical analysis should

[10] https://github.com/joaoarthurbm/designwizard

[11] End of Public Updates Notice. https://java.com/en/download/help/java_7.html

be conducted to capture different forms of entity update. Moreover, we may propose new data usage smells, which is not within the scope of this work.

4.4. Remarks on Implementation

Exclusion of Controversial Smells. We allow users to exclude every smell in command-line parameter `-e` in case they do not perceive them as real problems, *e.g.*, the impact of missing no argument constructor is almost negligible.

Drawbacks of Static Analysis. Static analysis has its unavoidable drawback since run-time information is not available. To cope with them, we need to specifically implement solutions for every up-mentioned case to detect the application of third-party plugins. For example, we cover the usage of libraries such as Lombok[12] and Apache Commons[13] to generate `equals`, `hashCode`, and accessor (`getter, setter`) methods. We may not be able to detect similar usage if practitioners use other libraries.

Detecting Pagination Smell. We locate the `setMaxResults` or `setFirstResult` method calls in parent methods with HQL appearance, and we analyze if the code component that called the method defines any `Integer` or `Long` object whose name contains "page" or "limit". This complies with the sample code of [13], which may be impractical, and may be improved in future work.

4.5. Research Opportunities

Since RDBMS and software applications are different systems, there still exist gaps (also known as "impedance mismatch" [29]) between them. ORM aimed to address this problem, but they introduced new code quality issues and more complexity. For practitioners, the quality of ORM code and the appropriate application of ORM should always be considered to make more reasonable refactoring plans. For researchers, more empirical studies could be made to measure the impact of data persistence code quality to software maintainability and reliability. To these ends, the research opportunities brought by our tool include but not limited to:

- Evaluating ORM smells in large-scale datasets (e.g., GitHub Mirrors [30]) to study their occurrence, impact, and interactions with other code smells and architectural issues;

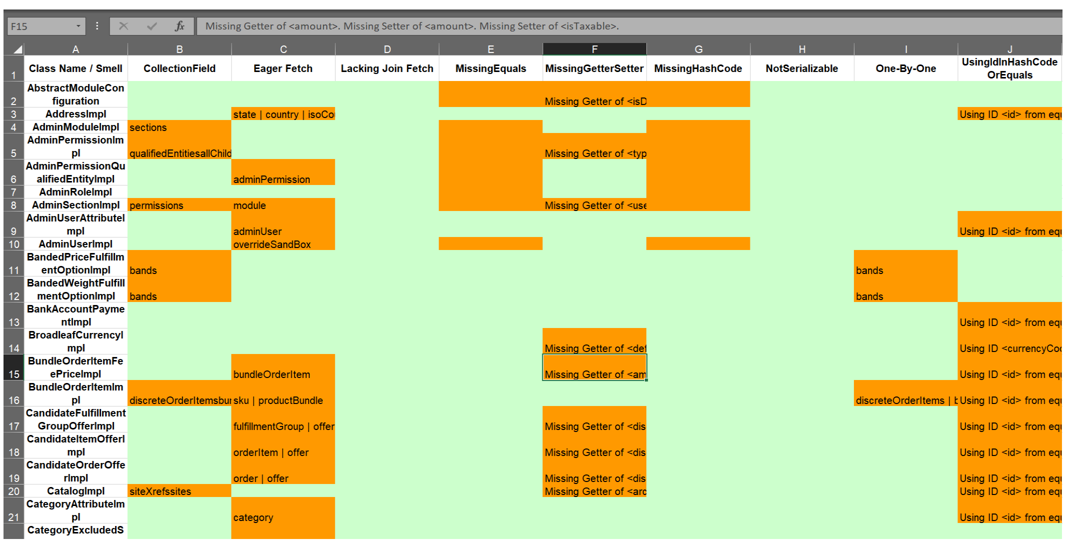
[12] https://projectlombok.org/

[13] https://commons.apache.org/

- Integrating our command-line tool into the process of Continuous Integration to generate reports after every development iteration of software code, or extending our code to build an IDE-based detection tool such as JDEODORANT [25] to provide just-in-time support for practitioners;
- Using our static analysis code base to develop detection methods for new ORM-related smells;
- Revealing the losses and gains of ORM detaching. For example, can we improve code maintainability and query performance by transferring to "lightweight" data persistence solutions such as semiautomatic ORM frameworks (*e.g.*, MyBatis[14]) and native SQL queries with manual data class mapping?

5. Illustrative Example

Figure 2 illustrates the exported `xls` report of the analyzed commercial project. Undetected smells are not presented. Fields in orange represents smelly, and texts in these fields are corresponding comments (*e.g.*, affected entity attributes). Light green fields refer to clean entities.



F15 | Missing Getter of <amount>. Missing Setter of <amount>. Missing Setter of <isTaxable>.

	A	B	C	D	E	F	G	H	I	J
1	Class Name / Smell	CollectionField	Eager Fetch	Lacking Join Fetch	MissingEquals	MissingGetterSetter	MissingHashCode	NotSerializable	One-By-One	UsingIdInHashCodeOrEquals
2	AbstractModuleConfiguration					Missing Getter of <isD				
3	AddressImpl		state | country | isCo							Using ID <id> from equ
4	AdminModuleImpl	sections								
5	AdminPermissionImpl	qualifiedEntitiesallChild				Missing Getter of <typ				
6	AdminPermissionQualifiedEntityImpl		adminPermission							
7	AdminRoleImpl									
8	AdminSectionImpl	permissions	module			Missing Getter of <use				
9	AdminUserAttributeImpl		adminUser							Using ID <id> from equ
10	AdminUserImpl		overrideSandBox							
11	BandedPriceFulfillmentOptionImpl	bands							bands	
12	BandedWeightFulfillmentOptionImpl	bands							bands	
13	BankAccountPaymentImpl									Using ID <id> from equ
14	BroadleafCurrencyImpl					Missing Getter of <def				Using ID <currencyCod
15	BundleOrderItemFeePriceImpl		bundleOrderItem			Missing Getter of <am				Using ID <id> from equ
16	BundleOrderItemImpl	discreteOrderItemsbu	sku | productBundle						discreteOrderItems | b	Using ID <id> from equ
17	CandidateFulfillmentGroupOfferImpl		fulfillmentGroup | offer			Missing Getter of <dis				Using ID <id> from equ
18	CandidateItemOfferImpl		orderItem | offer			Missing Getter of <dis				Using ID <id> from equ
19	CandidateOrderOfferImpl		order | offer			Missing Getter of <dis				Using ID <id> from equ
20	CatalogImpl	siteXrefssites				Missing Getter of <arc				Using ID <id> from equ
21	CategoryAttributeImpl		category							Using ID <id> from equ
	CategoryExcludedS									

Figure 2: A snapshot of the generated EXCEL report for the Broadleaf Commerce project.

[14] https://mybatis.org/mybatis-3/

Listing 3 is an example of the JSON output of HBSniff. The available information include names of smells (the name field), file paths (the file field), names of the classes (the className field), positions of the detected smells in the source code (the position field), and comments of the smells (the comment field). Such data is also available in the csv output.

Listing 3: Example of The JSON output for the portal project

```
{
    "Article.java": [
      {
        "name": "UsingIdInHashCodeOrEquals",
        "file": "...", // File Path
        "position": "(line 36,col 1)-(line 193,col 1)",
        "className": "Article",
        "comment": "Using ID <id> from equals.
        Using ID <id> from hashCode. "
      },

      ... // Other Smells
   ],

   "Other Hibernate Entities":[
      ... // Other Smells
   ], ...
}
```

6. Conclusions and Future Work

We presented a static analysis-based Java Hibernate ORM code smell detection tool called HBSniff which is capable for evaluating 14 smells and 4 mapping metrics in uncompiled Java project source codes. Moreover, we conducted unit tests and manual verification for the detectors and metrics to ensure the reliability of our implementation. We also evaluated the impact of the implemented performance smells, and we suggested refactoring them as soon as possible since most of them could greatly impact performance.

Future work includes: (1) proposing new ORM smells and improve the existing implementations, (2) extending our scope to Python ORMs, and (3) assessing the impact of ORM smells to architecture degradation and software maintainability.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61772200, and the Natural Science Foundation of Shanghai under Grant No. 21ZR1416300.

References

[1] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, The scent of a smell: An extensive comparison between textual and structural smells, IEEE Transactions on Software Engineering 44 (10) (2018) 977–1000.

[2] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation, Empirical Software Engineering 23 (3) (2018) 1188–1221.

[3] T.-H. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, P. Flora, An empirical study on the practice of maintaining object-relational mapping code in Java systems, in: Proc. 13th International Conference on Mining Software Repositories (MSR), 2016, p. 165–176.

[4] G. Vial, Lessons in persisting object data using object-relational mapping, IEEE Software 36 (6) (2019) 43–52.

[5] C. Nagy, A. Cleve, A static code smell detector for SQL queries embedded in Java code, in: Proc. IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017, pp. 147–152.

[6] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, G. Antoniol, On the prevalence, impact, and evolution of SQL code smells in data-intensive systems, in: Proc. of the 17th International Conference on Mining Software Repositories (MSR), 2020, p. 327–338.

[7] F. Gonçalves de Almeida Filho, A. D. Forte Martins, T. da Silva Vinuto, J. M. Monteiro, Í. Pereira de Sousa, J. de Castro Machado, L. Souza Rocha, Prevalence of bad smells in PL/SQL projects, in: Proc. IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 116–121.

[8] A. Torres, R. Galante, M. S. Pimenta, A. J. B. Martins, Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design, Information Software Technology 82 (2017) 1–18.

[9] M. Lorenz, J.-P. Rudolph, G. Hesse, M. Uflacker, H. Plattner, Object-relational mapping revisited: A quantitative study on the impact of database technology on O/R mapping strategies, in: Proc. 50th Hawaii International Conference on System Sciences (HICSS), 2017, pp. 4877–4886.

[10] S. Holder, J. Buchan, S. G. MacDonell, Towards a metrics suite for object-relational mappings, in: Proc. 1st International Workshop on Model-Based Software and Data Integration (MBSDI), 2008, pp. 43–54.

[11] L. Meurice, C. Nagy, A. Cleve, Detecting and preventing program inconsistencies under database schema evolution, in: Proc. IEEE 16th International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 262–273.

[12] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, P. Flora, Detecting performance anti-patterns for applications developed using object-relational mapping, in: Proc. 36th International Conference on Software Engineering (ICSE), 2014, pp. 1001–1012.

[13] S. Loli, L. Teixeira, B. Cartaxo, A catalog of object-relational mapping code smells for Java, in: Proc. 34th Brazilian Symposium on Software Engineering (SBES), 2020, pp. 82–91.

[14] T. M. Silva, D. Serey, J. C. A. de Figueiredo, J. Brunet, Automated design tests to check hibernate design recommendations, in: Proc. 33th Brazilian Symposium on Software Engineering (SBES), 2019, pp. 94–103.

[15] V. Lenarduzzi, V. Nikkola, N. Saarimäki, D. Taibi, Does code quality affect pull request acceptance? An empirical study, Journal of Systems and Software 171 (2021) 110806.

[16] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, There and back again: Can you compile that snapshot?, Journal of Software: Evolution and Process 29 (4) (2017) e1838.

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

[18] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora, Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks, IEEE Transactions on Software Engineering 42 (12) (2016) 1148–1161.

[19] P. Węgrzynowicz, Performance antipatterns of one to many association in hibernate, in: Proc. 2013 Federated Conference on Computer Science and Information Systems (FedCSIS), 2013, pp. 1475–1481.

[20] L. Meurice, C. Nagy, A. Cleve, Static analysis of dynamic database usage in Java systems, in: Proc. 28th International Conference on Advanced Information Systems Engineering (CAiSE), pp. 491–506.

[21] G. Procaccianti, P. Lago, W. Diesveld, Energy efficiency of ORM approaches: An empirical evaluation, in: Proc. 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, 2016, pp. 36:1–36:10.

[22] T.-H. Chen, W. Shang, A. E. Hassan, M. N. Nasser, P. Flora, Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications, in: Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 666–677.

[23] R. Singh, C. Bezemer, W. Shang, A. E. Hassan, Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm, in: Proc. 7th ACM/SPEC International Conference on Performance Engineering (ICPE), 2016, pp. 309–320.

[24] M. F. C. Nazário, E. Guerra, R. Bonifácio, G. Pinto, Detecting and reporting object-relational mapping problems: An industrial report, in: Proc. ACM/IEEE 13th International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019, pp. 1–6.

[25] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Ten years of JDeodorant: Lessons learned from the hunt for smells, in: Proc. IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 4–14.

[26] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, A. Serebrenik, Beyond technical aspects: How do community smells influence the intensity of code smells?, IEEE Transactions on Software Engineering 47 (1) (2021) 108–129.

[27] M. F. Aniche, G. Bavota, C. Treude, M. A. Gerosa, A. van Deursen, Code smells for model-view-controller architectures, Empirical Software Engineering 23 (4) (2018) 2121–2157.

[28] C. Nagy, L. Meurice, A. Cleve, Where was this SQL query executed? A static concept location approach, in: Proc. IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 580–584.

[29] W. R. Cook, R. Greene, P. Linskey, E. Meijer, K. Rugg, C. Russell, B. Walker, C. Wittig, Objects and databases: State of the union in 2006, in: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2006, p. 926–928.

[30] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: Ultra-large-scale software repository and source-code mining, ACM Transactions on Software Engineering and Methodology 25 (1) (2015) 7:1–7:34.

Required Metadata

Current executable software version

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	v1.6.8
S2	Permanent link to executables of this version	*https://github.com/HBSniff/HBSniff/releases/tag/v*1.6.8
S3	Legal Software License	GPL
S4	Computing platform/Operating System	Linux, OS X, Microsoft Windows.
S5	Installation requirements & dependencies	JDK 8.0
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	https://hbsniff.github.io/
S7	Support email for questions	hzj@mail.ecust.edu.cn

Table 3: Software metadata (optional)

Current code version

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	v1.6.8
C2	Permanent link to code/repository used of this code version	$https : //github.com/ HBSniff/HBSniff$
C3	Legal Code License	GPL
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Java
C6	Compilation requirements, operating environments & dependencies	JDK 8.0, Maven 5
C7	If available Link to developer documentation/manual	https://hbsniff.github.io/
C8	Support email for questions	hzj@mail.ecust.edu.cn

Table 4: Code metadata (mandatory)