An Empirical Study on the Practice of Maintaining Object-Relational Mapping Code in Java Systems

Tse-Hsun Chen
Queen's University
Ontario, Canada
tsehsun@cs.queensu.ca

Weiyi Shang
Concordia University
Quebec, Canada
shang@encs.concordia.ca

Jinqiu Yang
University of Waterloo
Ontario, Canada
j223yang@uwaterloo.ca

Ahmed E. Hassan
Queen's University
Ontario, Canada
ahmed@cs.queensu.ca

Michael W. Godfrey
University of Waterloo
Ontario, Canada
migod@uwaterloo.ca

Mohamed Nasser
BlackBerry
Ontario, Canada

Parminder Flora
BlackBerry
Ontario, Canada

ABSTRACT

Databases have become one of the most important components in modern software systems. For example, web services, cloud computing systems, and online transaction processing systems all rely heavily on databases. To abstract the complexity of accessing a database, developers make use of Object-Relational Mapping (ORM) frameworks. ORM frameworks provide an abstraction layer between the application logic and the underlying database. Such abstraction layer automatically maps objects in Object-Oriented Languages to database records, which significantly reduces the amount of boilerplate code that needs to be written.

Despite the advantages of using ORM frameworks, we observe several difficulties in maintaining ORM code (i.e., code that makes use of ORM frameworks) when cooperating with our industrial partner. After conducting studies on other open source systems, we find that such difficulties are common in other Java systems. Our study finds that i) ORM cannot completely encapsulate database accesses in objects or abstract the underlying database technology, thus making ORM code changes more scattered; ii) ORM code changes are more frequent than regular code, but there is a lack of tools that help developers verify ORM code at compilation time; iii) we find that changes to ORM code are more commonly due to performance or security reasons; however, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems. Our study highlights the hidden maintenance costs of using ORM frameworks, and provides some initial insights about potential approaches to help maintain ORM code. Future studies should carefully examine ORM code, in particular given the rising use of ORM in modern software systems.

1. INTRODUCTION

Managing data consistency between source code and database is a difficult task, especially for complex large-scale systems. As more systems become heavily dependent on databases, it is important to abstract the database accesses from developers. Hence, developers nowadays commonly make use of Object-Relation Mapping (ORM) frameworks to provide a conceptual abstraction between objects in Object-Oriented Languages and data records in the underlying database. Using ORM frameworks, changes to object states are automatically propagated to the corresponding database records.

A recent survey [54] shows that 67.5% of Java developers use ORM frameworks (i.e., Hibernate [10]) to access the database, instead of using JDBC. However, despite ORM's popularity and simplicity, maintaining ORM code (i.e., code that makes use of ORM frameworks) may be very different from maintaining regular code due to the nature of ORM code. Prior studies [19, 11, 45] usually focus on the evolution and maintenance of database schemas. However, the maintenance of database access code, such as ORM code, is rarely studied. In particular, since ORM introduces another abstraction layer on top of SQL, introducing extra burden for developers to understand the exact behaviour of ORM code [7], maintaining ORM code can be effort consuming.

During a recent cooperation with one of our industrial partners, we examined the challenges associated with maintaining a large-scale Java software system that uses ORM frameworks to abstract database accesses. We observed several difficulties in maintaining ORM code in Java systems. For example, changes that involve ORM code are often scattered across many components of the system.

With such observations on the industrial system, we sought to study several open source Java systems that heavily depend on ORM in order to verify whether maintaining ORM code in these systems also suffers from the difficulties that are observed in the industrial system. We conducted an empirical study on three open source Java systems, in addition to the one large-scale industrial Java system. We find that the difficulties of maintaining ORM code are common among the studied systems, which further highlights that the challenges of maintaining ORM code is a wide ranging concern.

In particular, we investigate the difficulties of maintaining ORM code through exploring the following research questions:

RQ1: ***How Localized are ORM Code Changes?***
We find that code changes that involve ORM code are more scattered and complex, *even after we control for fan-in* (statistically significant). In other words, ORM fails to completely encapsulate the underlying database accesses in objects; hence, making ORM code harder to maintain compared to regular code.

RQ2: ***How does ORM Code Change?***
We find that ORM code is changed more frequently (115%–179% more) than non-ORM code. In particular, ORM model and query code are often changed, which may increase potential maintenance problems due to lack of query return type checking at compilation time (hence many problems might remain undetected till runtime in the field). On the other hand, developers do not often tune ORM configurations for better performance. Therefore, developers may benefit from tools that can automatically verify ORM code changes or tune ORM configurations.

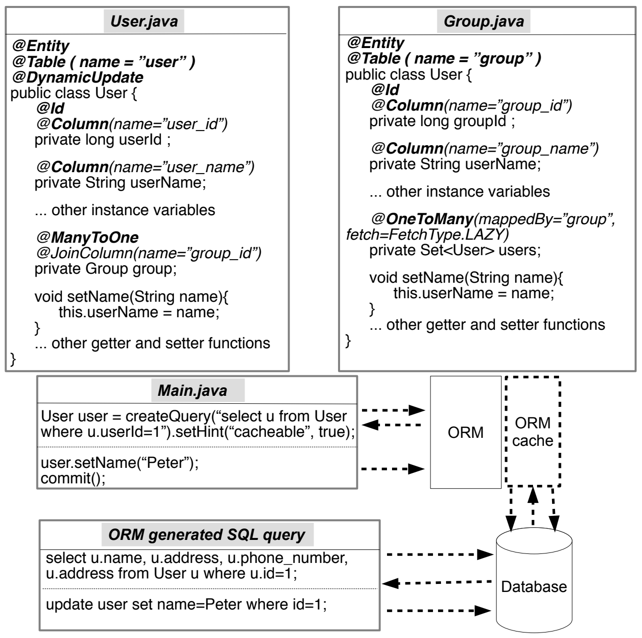
RQ3: ***Why are ORM Code Changed?***
Through a manual analysis of ORM code changes, we find that compared to regular code, ORM code is more likely changed due to performance and security reasons. However, since ORM code is syntactically (i.e., have a unique set of APIs, and different code structure/grammar) and semantically (i.e., access databases) different from regular code, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems.

Our study highlights that practitioners need to be aware of the maintenance cost when taking advantage of the conveniences of ORM frameworks. Our study also provides some initial insights about potential approaches to help reduce the maintenance effort of ORM code. In particular, our results helped our industrial partner understand that some types of problems may be more common in ORM code, and what kinds of specialized tools may be needed to reduce the maintenance effort of ORM code.

Paper Organization. Section 2 briefly introduces different ORM frameworks, discusses how ORM frameworks may translate object manipulations to SQL queries, shows different types of ORM code, and discusses the use of ORM in practice. Section 3 conducts a preliminary study on the evolution of different types of ORM code and ORM code density. Section 4 presents the results of our research questions. Section 5 discusses the implications of our findings. Section 6 discusses the potential threats to the validity of our study. Section 7 surveys related studies on database and software evolution. Finally, Section 8 concludes the paper.

2. BACKGROUND OF OBJECT-RELATIONAL MAPPING

This section provides some background knowledge of ORM.



User.java
```
@Entity
@Table ( name = "user" )
@DynamicUpdate
public class User {
    @Id
    @Column(name="user_id")
    private long userId ;

    @Column(name="user_name")
    private String userName;

    ... other instance variables

    @ManyToOne
    @JoinColumn(name="group_id")
    private Group group;

    void setName(String name){
        this.userName = name;
    }
    ... other getter and setter functions
}
```

Group.java
```
@Entity
@Table ( name = "group" )
public class User {
    @Id
    @Column(name="group_id")
    private long groupId ;

    @Column(name="group_name")
    private String userName;

    ... other instance variables

    @OneToMany(mappedBy="group",
fetch=FetchType.LAZY)
    private Set<User> users;

    void setName(String name){
        this.userName = name;
    }
    ... other getter and setter functions
}
```

Main.java
```
User user = createQuery("select u from User
where u.userId=1").setHint("cacheable", true);

user.setName("Peter");
commit();
```

ORM generated SQL query
```
select u.name, u.address, u.phone_number,
u.address from User u where u.id=1;

update user set name=Peter where id=1;
```

Figure 1: Example of database entity classes and how ORM translates object manipulation to SQL queries.

We first provide a brief discussion of how ORM frameworks are used in practice. Then, we give a simple example of how to access a database using ORM. Finally, we discuss the different types of ORM code.

2.1 ORM Frameworks in Practice

ORM is widely used by developers due to the simplicity of the conceptual abstraction that it provides between the source code and the underlying database [33]. A recent survey [54] shows that 67.5% of Java developers use ORM frameworks (i.e., Hibernate [10]) to access the database, instead of using JDBC. Modern programming languages, such as Java, C#, Python, and Ruby, all provide ORM solutions. Java, in particular, provides a standard API for ORM, which is called the Java Persistent API (JPA).

There exist many implementations of JPA, such as Hibernate [10], OpenJPA [16], EclipseLink [17], and IBM WebSphere (which uses a modified version of OpenJPA) [30]. These implementations share similar designs and functionalities, although they have implementation-specific differences (e.g., varying performance [38]). Developers can switch, in theory, between different JPA implementations with minimal changes. In this paper, we focus our ORM study on JPA due to its popularity.

2.2 Accessing a Database Using ORM

When using an ORM framework, developers usually use annotations to map a Java class as a database entity class (i.e., the class is mapped to a database table). Although different ORM frameworks may have different mapping mechanisms, the main concepts are common. Using ORM, developers work directly with objects, and ORM translates operations on objects to SQL queries. Figure 1 provides an example to illustrate how ORM does such mappings and translations. Group and User are two database entity classes that are mapped to database tables. Main.java sends a request to the database to retrieve a user row from the database, and to store the row as a user object. Then, Main.java updates data in the user object, and the corresponding change is automatically propagated to the database without developer's intervention.

2.3 Types of ORM code

We consider ORM code as the code that contains certain ORM API calls or annotations. There are different types of ORM code, which are distinguishable from ordinary Java code. The ORM code can be classified into the following three types:

1) Data Model and Mapping. ORM uses **@Entity** to declare a class as a database entity class, and specifies the database table to which the class is mapped using **@Table**. Each instance of the database entity object is mapped to a row in the database. For example in Figure 1, the User class is mapped to the user table in the database, and the Group class is mapped to the group table in the database. **@Column** maps instance variables to columns in the database. For example, userName is mapped to the column user_name in the user table. Developers can also specify relationships among different database entity classes. There are four different relationships: **@OneToOne**, **@OneToMany**, **@ManyToOne**, and **@ManyToMany**. For example in Figure 1, we specify a **@ManyToOne** relationship between the User class and the Group class (and a **OneToMany** relationship between Group and User). Developers can also specify how associated entities should be retrieved from the database. A fetch type of **EAGER** means that the associated entity (User) will be retrieved once the owner entity is retrieved (Group). A fetch type of **LAZY** means that the associated entity will be retrieved only when it is needed in the code. We expect ORM data model code will be changed most frequently, since database schemas may change as systems evolve [45].

2) Performance Configurations. Developers can use cache configurations to reduce the number of calls to the database. For example in Main.java, we configure the resulting SQL query (*setHint("cacheable", true)*) to be stored in the ORM cache. ORM provides other configurations to help optimize performance. For example, developers may configure ORM to update only modified columns for rows in the database (**@DynamicUpdate**). Without using **@DynamicUpdate**, the ORM generated SQL queries will contain all columns of an entity object, even though the values are not updated. Using **@DynamicUpdate** can help reduce data transmission overheads and improve system performance. We expect changes to the ORM performance configuration code overtime, because performance tuning is likely to be done when features are modified or introduced [52, 13].

3) ORM Query Calls. ORM provides APIs for developers to retrieve objects from the database using the primary key (e.g., User u = find(User.class, ID)). In some situations where developers require more complex select criteria, developers can use the ORM query calls. ORM query calls are similar to SQL queries, where both languages allow developers to do customized selections. ORM query calls only support select, because ORM does the updates automatically. Main.java in Figure 1 shows an example of an ORM query call for selecting user from the database. Such ORM query calls are seen as a violation of the conceptual abstraction offered by ORM frameworks. In addition, these query calls do not have type checking (e.g., the existence of the queried columns is not checked during compilation), and must be maintained carefully, otherwise problems arise when database schema changes occur. We expect ORM query code to exhibit changes overtime, since adding or modifying features is likely to require changing ORM query code.

Table 1: Statistics of the studied systems in the latest version. ES is not shown in detail due to NDA.

	Lines of code (K)	No. of Java files	% files contain ORM code	No. of studied versions	Latest studied version	Median ORM code density among all all versions
Broadleaf	363K	2,249	13%	79	3.1.0	1.3%
Devproof	53.7K	541	20%	7	1.1.1	0.7%
JeeSite	397K	126	16%	5	1.0.4	1.1%
ES	>300K	>3,000	4%	>10	—	< 1%

2.4 Identifying ORM Code

We developed a Java code analyzer that looks for ORM API calls. Our code analyzer classifies the ORM code into the three above-mentioned types.

3. PRELIMINARY STUDY

In this section, we first introduce our studied systems, then we discuss the evolution of ORM code in these systems.

3.1 Studied Systems

We study three open-source systems (Broadleaf Commerce [9], Devproof Portal [44], and JeeSite [51]) and one large-scale industrial system (ES). Due to a Non-Disclosure Agreement (NDA), we cannot expose all the details of ES. Table 1 shows an overview of the studied systems, as well as the overall ORM code density. All of the studied systems follow the typical Model-View-Controller (MVC) design [36], and use Hibernate as the implementation of ORM. Broadleaf Commerce is an e-commerce system, which is widely used in both open-source and commercial settings. Devproof Portal is a fully featured portal, which provides features such as blogging, article writing, and bookmarking. JeeSite provides a framework for managing enterprise information, and offers a Content Management System (CMS) and administration platform. ES is a real-world industrial system that is used by millions of users worldwide on a daily basis.

3.2 Evolution of ORM Code

We first conduct a preliminary study on the evolution of ORM code. We use the following metrics to study the evolution of ORM code:

- Number of database table mappings;
- Number of ORM query calls;
- Number of performance configurations;
- ORM code density.

We define ORM code density as the total number of ORM code (i.e., lines of code that perform database table mapping, performance configuration calls, and ORM query calls) divided by the total lines of code. Below we discuss our findings for the above-mentioned metrics.

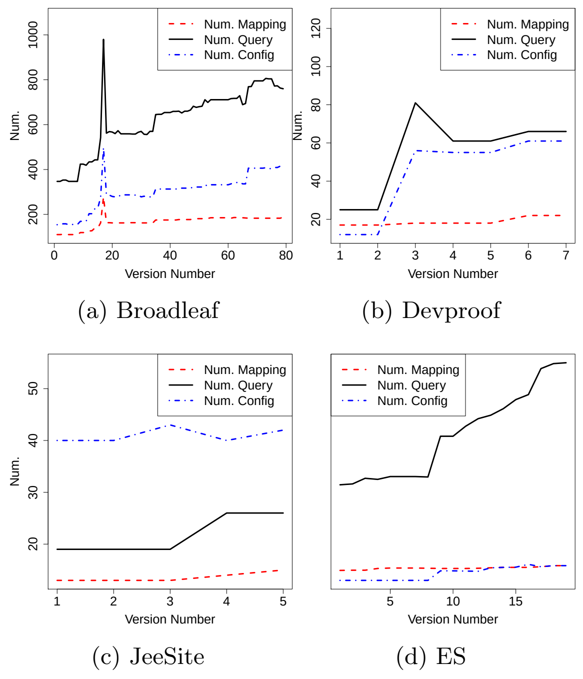


Figure 2: Evolution of the total number of database table mappings, ORM query calls, and ORM configurations. The values on the y-axis are not shown for ES due to NDA.

Figure 2 shows the evolution of the three types of ORM code. We find that, in general, the number of ORM query calls has the steepest increase overtime. We also find that the number of database table-mappings does not change much overtime, and that the total number of ORM performance configurations remains relatively stable in JeeSite and ES. We also find that the change rate of ORM configuration code is lower than the other types of ORM code in some systems. Since ORM configuration code is usually applied on ORM queries and ORM table mapping code, this finding may indicate that not all developers spend enough time tuning or adding the configurations, which may result in performance problems in ORM-based systems [7].

4. CASE STUDY RESULTS

We now present the results of our research questions. Each research question is composed of four parts: motivation, approach, experimental results, and discussion.

RQ1: How Localized are ORM Code Changes?

Motivation. To verify the generalizability of our observation in ES and examine if ORM code changes are more scattered (i.e., complex) by nature, we study the complexity of ORM code changes in this RQ.

Approach. We study the code change complexity after normalizing the fan-in of ORM code. High degree of dependence (i.e., high fan-in) is likely to lead to higher maintenance costs (due to changing more dependent files). Hence, by controlling for fan-in, we ensure that our obverstaions are more likely due to the nature of ORM code rather than its high fan-in.

Dependence on the Files that Contain ORM Code. To measure the dependence on the files that contain ORM code, we compute the degree of fan-in at the file level [26]. We annotate each file as one that contains ORM code (ORM file) or one that does not contain ORM code (non-ORM file). Fan-in measures the number of files that depend on a given file. For example, if file B and C both are calling one or more functions in file A, then the degree of fan-in of file A is two. We compute the fan-in metric for all files for all studied versions, and then we compare the degree of fan-in between ORM and non-ORM files.

Complexity of ORM Code Change. To measure the complexity of ORM code changes, we compute the following metrics for each commit:

- Total number of code churn (lines inserted/deleted) in a commit;
- Total number of files that are modified in a commit;
- Commit change entropy [24].

The three above-mentioned metrics are used in prior studies to approximate the complexity of code changes in a commit [53, 50, 4]. We classify commits into ORM commits (i.e., commits that modify ORM code), and non-ORM commits, and compare the metrics between these two types of commits.

Entropy measures the uncertainty in a random variable, and maximum entropy is achieved when all the files in a commit have the same number of modified lines. In contrast, minimum entropy is achieved when only one file has the total number of modified lines in a commit. Therefore, higher entropy values represent a more complex change (i.e., scattered changes), where smaller entropy values represent a less complex change (i.e., changes are concentrated in a small number of files) [24].

To measure the change entropy, we implement the normalized Shannon Entropy to measure the complexity of commits [24, 53]. The entropy is defined as:

$$H(Commit) = \frac{-\sum_{i=1}^{n} p(File_i) * log_e p(File_i)}{log_e(n)}, \qquad (1)$$

where n represents the total number of files in a commit $Commit$, and $H(Commit)$ is the entropy value of the commit. $p(File_i)$ is defined as the number of lines changed in $File_i$ over the total number of lines changed in every file of that commit. For example, if we modify three files A (modify 1 line), B (modify 1 line), and C (modify 3 lines), then p(A) will be $\frac{1}{5}$ (i.e., $\frac{1}{1+1+3}$).

Statistical Tests for Metrics Comparison. To compare the metric values between ORM and non-ORM files, we use the single-sided Wilcoxon rank-sum test (also called Mann-Whitney U test). We choose the Wilcoxon rank-sum test over Student's t-test because our metrics are skewed, and the Wilcoxon rank-sum test is a non-parametric test, which does not put any assumption on the distribution of two populations. The Wilcoxon rank-sum test gives a p-value as the test outcome. A p-value ≤ 0.05 means that the result is statistically significant, and we may reject the null hypothesis (i.e., the two populations are different). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us if one population is statistically significantly larger than the other. In this RQ, we set the alternative hypothesis to check whether the metrics for ORM commits are *larger* than that of non-ORM commits.

Prior studies have shown that reporting only the p-value may lead to inaccurate interpretation of the difference between two populations [41, 34]. When the size of the populations is large, the p-value will be significant even if the difference is very small. Thus, we report the effect size (i.e., how large the difference is) using *Cliff's Delta* [8]. The strength of the effects and the corresponding range of *Cliff's Delta* values are [46]:

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } \textit{Cliff's Delta} < 0.147 \\ \text{small} & \text{if } 0.147 \le \textit{Cliff's Delta} < 0.33 \\ \text{medium} & \text{if } 0.33 \le \textit{Cliff's Delta} < 0.474 \\ \text{large} & \text{otherwise} \end{cases}$$

Results. ***We find that files that contain ORM code have a higher degree of fan-in (statistically significant) in the studied systems, which make them good candidate systems for our study.*** We compute the degree of fan-in for each version separately, and report the p-value of comparing the degrees of dependency from ORM and non-ORM code of each version. The p-value is statistically significant (<0.05) in every version of all the studied systems (ORM files have a higher fan-in). Table 2 shows the median of the effect sizes across all versions of the degree of fan-in of files that contain ORM code and files that do not contain ORM code. The effect sizes of the difference are non-trivial in all of our studied systems. These findings highlight and confirm the central role of ORM code in the studied software systems and their evolution. Thus, these systems are indeed good candidates for our study.

ORM-related commits modify more lines of code and files, and the commits are more scattered, even after we control for fan-in. We obtain a p-value of << 0.001 for the result of the Wilcoxon rank-sum test for the total code churn, the total files modified, and the change entropy of a commit in the studied systems. Our findings indicate that commits that modify ORM code are more complex (statistically significant) than commits that do not modify ORM code. From Table 3 we can also see that the median of the metrics for commits that modify ORM code are all larger than commits that do not modify ORM code. We find that 88% of the median effect sizes of the ORM code complexity is at least medium, which further supports the result of our Wilcoxon rank-sum test.

Since we find that files with ORM code have a higher fan-in in general, such characteristics may affect the complexity of changes that involve ORM code. As a result, we further study the complexity of ORM code changes after controlling for fan-in. We calculate the total fan-in of all the files in each commit, and then we normalize the commit complexity by dividing it by the total fan-in of all involved files. For example, if a commit modifies 1,000 lines of code, and the total fan-in of all the files that are modified in the commit is 100, the normalized total lines of code modified is 10 (1,000/100). We find that, after controlling for fan-in, commits that modify ORM code are still more complex (all statistically significant, except for change entropy in JeeSite).

It may first seem expected that, since ORM code is one of the core components of a system (i.e., has higher fan-in), ORM code changes should be more scattered. However, the finding highlights a potential issue with ORM code. Although the goal of ORM code is primarily on abstracting relational databases in object-oriented programming languages, we find that ***the underlying database is not completely encapsulated inside objects, so changing ORM code requires changing many other files***. Further studies are needed to better understand and resolve the failure of ORM code in keeping database knowledge encapsulated within objects.

Table 2: Medians (*Med.*, computed across all versions) and effect sizes (*Eff.*, median across all versions) of fan-in of ORM and non-ORM files. All differences are statistically significant. We only show the effect sizes for ES due to NDA.

Metric	Type	Broadleaf Med.	Broadleaf Eff.	Devproof Med.	Devproof Eff.	JeeSite Med.	JeeSite Eff.	ES Eff.
Fan-in	ORM	11.0	0.29	5.9	0.32	6.2	0.81	0.34
	Non-ORM	6.8		3		2		

Table 3: Medians (*Med.*, computed across all versions) and effect sizes (*Eff.*, averaged across all versions) of the complexity of ORM code changes. All differences are statistically significant. We only show the effect sizes for ES due to NDA.

Metric	Type	Broadleaf Med.	Broadleaf Eff.	Devproof Med.	Devproof Eff.	JeeSite Med.	JeeSite Eff.	ES Eff.
LOC modified	ORM	102	0.44	241	0.60	773	0.76	0.50
	Non-ORM	17		30		21		
Files modified	ORM	6	0.53	11	0.51	13	0.69	0.53
	Non-ORM	2		3		2		
Entropy	ORM	0.78	0.34	0.80	0.29	0.85	0.37	0.28
	Non-ORM	0.00		0.59		0.61		

Discussion. We do not know whether the high fan-in of ORM files is caused by the design of ORM code, or simply because these files are the core components of the studied systems. Thus, we conduct an experiment on the degree of fan-in of files with/without ORM code. We first find the top 100 files with the highest fan-in values for each studied system, and examine how many of the top 100 files have ORM code (Table 4). We find that in all the studied systems, only about 11–35 files among the top 100 files contain ORM code. In short, we find that files with ORM code are not the only core components of a system; nevertheless files with ORM code tend to have a higher change complexity.

Our finding helps our industrial partner recognize the high scatteredness of ORM code changes. ORM code changes are considered much riskier, and require careful attention and reviewing. Our findings are consistent among the studied open source systems, and the problems are not specific to ES. Further studies are needed to understand the reasons that ORM code fail to completely encapsulate the underlying database concerns within objects.

Table 4: Number of ORM files in the top 100 files with the largest degree of fan-in (averaged across versions).

Metric	Broadleaf	Devproof	JeeSite	ES
# of files with ORM code in the top 100	11.5	35	18	14.3

Table 5: Percentage of files that contain each type of ORM code in the top 100 high fan-in files.

Type	Broadleaf	Devproof	JeeSite	ES
Data Model	7%	8%	17%	16%
ORM Query Call	1%	26%	3%	1%
Perf. Config. Call	3%	8%	15%	10%

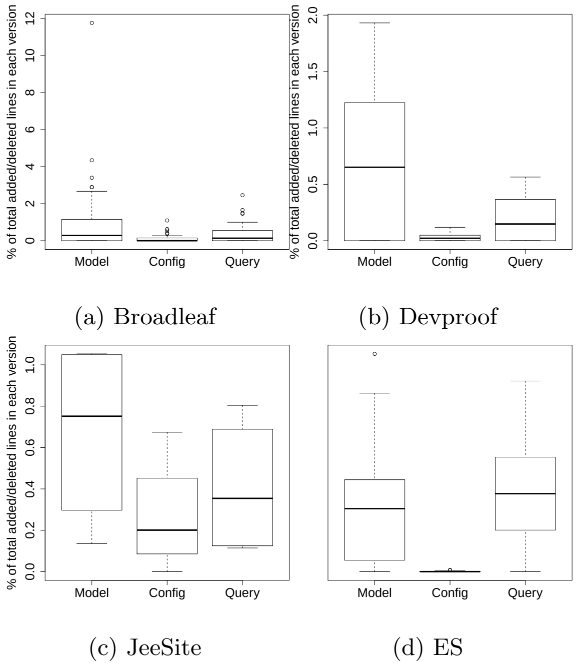


Figure 3: Distribution of the percentage of code churn for different types of ORM code changes across all the studied versions. We omit the scale for ES due to NDA.

> *We find that ORM cannot completely encapsulate the underlying database accesses in objects, making ORM code changes more scattered.*

RQ2: How does ORM Code Change?

Motivation. In RQ1, we found that changes involve ORM code are usually more complex and more scattered. In this RQ, we want to further study the change frequency of each type of ORM code to find out which type of ORM code requires the most maintenance effort. Namely, we study how developers change different types of ORM code (i.e., data model, ORM query, and performance configuration).

Approach. To answer this RQ, we compute the total code churn (i.e., added and deleted lines) and ORM code churn (i.e., added and deleted ORM code) between versions. We are particularly interested in how developers change different types of ORM code, since these changes directly reflect the maintenance activity on ORM code. Therefore, we compute the following metrics:

Table 6: Total code churn and ORM-related code churn in the studied systems.

	Code Churn	ORM Churn	$churn_{\text{model}}$	$churn_{\text{config}}$	$churn_{\text{query}}$
Broadleaf	1472K	22K (2%)	67%	7%	26%
Devproof	88K	1.1K (1%)	68%	5%	27%
JeeSite	11K	194 (2%)	49%	18%	33%
ES	—	<1%	41%	0.2%	59%

- Total code churn;
- Total ORM code churn;
- Code churn for ORM data model, query calls, and performance configurations.

ORM data model code churn ($churn_{\text{model}}$) computes the amount of churn related to data model code. ORM query-related code churn ($churn_{\text{query}}$) measures the amount of ORM query-related churns. ORM performance configuration code churn ($churn_{\text{config}}$) measures the amount of ORM performance configurations churns (e.g., cache configurations). We also measure the total amount of code churn and ORM code churn for comparison.

We use the following metric to measure the churn ratio of ORM code:

$$churn_{\text{ratio}} = \frac{\text{ORM code churn}}{\text{ORM code density}}, \qquad (2)$$

where a high $churn_{\text{ratio}}$ value means that ORM code has a higher chance of being modified (assuming each line of code has the same probability of being modified).

Results. ***Tools such as type checker for ORM queries and automated configuration tuning may help developers with ORM code maintenance.*** In Broadleaf, we find that about 2% (22K LOC) of total code churn (across all versions) is related to ORM (Table 6). We also find similar amount of ORM code churn in Devproof (1% of all churn) and JeeSite (2% of all churn). Given the low ORM code density in the entire system (Table 1), this implies that ORM code changes more frequently than that of other code. The $churn_{\text{ratio}}$ for Broadleaf, Devproof, JeeSite, and ES are 115%, 179%, 164%, and 120%, respectively.

In Broadleaf, Devproof, and JeeSite, most ORM code changes are related to data models. Developers sometimes make changes to the data models about how the classes are mapped to the database tables. Such changes may lead to other ORM code changes and refactorings. Consider the following example from the studied systems:

```
-@ManyToMany(fetch = FetchType.LAZY,
-targetEntity = OrderItemImpl.class)
-@JoinTable(name = "GIFTWRAP_ORDERITEM")
+@OneToMany(fetch = FetchType.LAZY,
+mappedBy = "giftWrapOrderItem",
+targetEntity = OrderItemImpl.class)
private List<OrderItem> wrappedItems =
new ArrayList<OrderItem>();
```

Developers change the relationship between OrderItemImpl and GitWrapOrderItem from **@ManyToMany** to **@OneToMany** due to data model changes and refactoring. Such

Table 7: Median churn percentage of each type of ORM code across all versions.

	Model	Config	Query
Broadleaf	0.9%	0.0%	1.5%
Devproof	45.6%	9.8%	56.8%
JeeSite	21.8%	17.8%	38.0%
ES	5.4%	0.0%	8.9%

changes may cause some other side effects such as the properties of the data model in the code (e.g., entity relationships) no longer matching the properties in the database [42].

We find that developers in all the studied systems also change ORM query calls very often. These ORM query calls provide developers a non-encapsulated (i.e., not Object-Oriented) way to retrieve data from the database. However, evolution of database models and frequent changes to these ORM query calls may cause some problems, since there exists no type checking for ORM query calls at compilation time [2]. For example, Nijjar *et al.* [42] found that there may exist problems in ORM data models due to the abstraction of the underlying relational models. ***Therefore, having tools that can help developers with compile time code verification or type checking for ORM query calls may reduce ORM code maintenance effort.***

Finally, we find that changes to ORM performance configurations are less frequent than the other types of ORM code. This finding is alarming, since the performance of ORM code is related to how ORM code is configured [7]. Prior studies [52, 13] from the database community have shown that tuning the performance of database-related code (e.g., SQL) is a continuous process, and needs to be done as systems evolve. ***As a result, automatically helping developers configure ORM code will be beneficial when maintaining ORM code.*** In recent work [5], we have followed up on this finding, and implement an automated tool for tuning ORM performance configuration code. We find that our tool can help improve system throughput by 27–138%.

Discussion. Since the amount of each type of ORM code is different, we normalize the churn by the total existence of each type of ORM code in the system. For each type of ORM code, we divide the number of modified lines of code by the total amount of such code. We compute such number for each version, and we report the median value in Table 7. We find that the change size of ORM query code is about 1.5%–56.8% of all ORM query code, which is significantly larger than the other two types of ORM code. We find a consistent trend in all studied systems. Our result shows that ORM query code is changed more often (after normalization), even though ORM query code bypass the ORM abstraction layer and may be error-prone (no type checking at compile time).

ORM code has less density but is changed more frequently (115%–179% more) than non-ORM code. We also find that both ORM query calls and models are changed frequently, while ORM configurations are rarely changed. Thus, developers may benefit from tools for verifying the type of returned objects by ORM queries, or tools for automated tuning of ORM performance configurations.

Table 8: Manually derived categories for commits.

Category	Description	Abbr.
Bug Fix	Code is modified to fix a bug	Bug
Compatibility Issue	Modifications to allow code to work in an other environment	Compat
Feature Enhancement	Enhance current functionalities	Enhance
New Feature	Add new functionalities	New
Performance Config.	Performance and performance configuration tuning	Config
Refactoring	Code refactoring	Refactor
Model	Database schema/ORM data model is changed	Model
Security Enhancement	Enhance security	Secure
Test	Add test code	Test
Upgrade	Upgrade dependent changes	Upgrade
GUI	Modified graphical/web user interface	GUI
Documentation	Updated documentation	Doc
Build	Modify/Update build files	Build

RQ3: Why are ORM Code Changed?

Motivation. In the previous RQs, we study the characteristics of different ORM code changes. However, the reasons for changing the ORM code are not known. Since ORM code is dependent on the database, the reasons for ORM code changes may be different from regular code changes. However, ORM code may also be different from regular SQL queries, since ORM abstracts SQL queries from developers. Thus, in this RQ, we manually study the reasons for ORM code changes, and we compare such reasons with regular code changes (i.e., changes that do not modify ORM code).

Approach. We manually study the reasons for the changes that developers make. We first collect all the commits, and we annotate each commit as ORM-related or regular commits (commits that do not modify ORM code). In total, there are 2,223 commits that change ORM code, and 9,637 commits that do not change ORM code (across all the studied open source systems). We do not show data from ES in this study due to NDA, but our findings in ES (e.g., the categories and the distributions) are very similar to what we found in the open source systems. In order to achieve a confidence level of 95% with a confidence interval of 5% in our results [3], we randomly sample 328 ORM-related commits and 369 non-ORM commits for our manual study. We first examine the randomly sampled commits (both ORM and non-ORM commits) with no particular categories in mind. Then, we manually derive the set of categories for which these commits belong. In total, we derive 13 categories for the commits. Table 8 has the descriptions for the categories. For commits that belong to multiple categories, we assign the commits to all the categories to which they belong. We then study how the studied commits are distributed in these categories.

Results. ***ORM code changes are more likely due to performance, compatibility, and security problems compared to regular code; automated techniques for detecting such problems in ORM-based systems may be beneficial.*** Figure 4 shows the distributions of the studied commits and the category to which they belong. We find that ORM and non-ORM code changes share some common reasons. Developers spend a large amount of effort on ORM code refactoring (30.64% of all commits that modify ORM code). In addition, 22.01% of the commits are related to bug fixing, and 20.33% of the commits are related to feature enhancement. In short, more than 70% of the commits that change ORM code are related to code maintenance activities (i.e., refactoring, enhancement, and bug fixing) [27]. We find that these maintenance activities have very similar distributions in non-ORM commits.

Nevertheless, some categories of ORM commits do not exist in non-ORM commits (i.e., *Compat*, *Config*, *Model*, and *Secure*). Our sampled commits show that these reasons are more likely to result in ORM code changes, although we expect that *Model* only exists in commits that contain ORM code. We also find that ORM frameworks play a central role in the performance of ORM-based systems – developers change ORM configuration code more frequently for performance improvement. In RQ2 we find that ORM configuration code is not frequently changed. However, in our prior study [7], we find that developers may not always be aware of the performance impact of the ORM code due to the database abstraction (i.e., developers may not know the code they write would result in slow database accesses). Hence, there may be many places in the code that require performance tuning, and tools that can automatically change/tune ORM configuration code can be beneficial to developers.

Moreover, even though ORM ideally should ensure that the code is database-independent (i.e., porting a system to a different database technology should require no code changes), we still find some counterexamples. Finally, since ORM code need to send user requests to the underlying database, security may also more likely to be a concern (e.g., SQL injection attacks). In the discussion, we further discuss some of the compatibility, performance, and security problems that we found in our manual study.

Discussion. Even though ORM is touted as a solution that would ensure that ORM code would work against all kinds of database technologies, we still observe database compatibility issues. For example, in Broadleaf, developers change the database table name due to an Oracle-specific size limit on table names.

During our manual study, the most commonly observed ORM performance configuration changes are due to data caching. Consider the following example from Broadleaf:

```
+@Cache(usage =
+CacheConcurrencyStrategy.READ_ONLY)
private Map<String, String> images;
```

The images variable stores a binary image for each category of items (represented using a Java String). The image for each category does not change often, yet they often have a large data size. Frequently retrieving the large binary data from the database may cause significant performance overheads and reduce user experience [49]. As a result, the developers cache the images into ORM cache (Figure 1). Since the images are read-only, adding a *read-only* cache significantly improves system performance. Note that although this problem may also exist in other systems, the problem may have higher prevalence in ORM-based systems. Since ORM does not know whether image data is needed in the code, ORM will always fetch the image data from the database under default configuration. This problem may be easily observed if developers manually write SQL queries and decide which columns should be retrieved from the database table.

Finally, we see that developers refactor how the database entity classes are designed and called to enhance system security. They do this by providing more validation rules (i.e., access control) to the user requests and to database entity object that are being retrieved from the database.

We find that although there are common reasons for ORM and non-ORM code changes, some problems are more likely to cause of ORM code changes. Future studies may propose different techniques to detect such problems in order to assist developers with maintaining ORM-based systems.

Based on our manually studied samples, we find that compatibility, performance, and security problems are more common to result in ORM code changes. Thus, developing tools to detect such problems in ORM code may help developers with maintaining ORM-based systems.

5. HIGHLIGHTS AND IMPLICATIONS OF OUR FINDINGS

Our study has helped our industry partner recognize some key challenges associated with maintaining ORM code. Our study on the open source systems confirms that our findings are not specific to ES, and maintaining ORM code may be a wide-ranging concern. Although ORM frameworks are widely used in industry, many ORM-specific problems do not have a solution from the research world.

The highlights and implications of our findings are:

- ***ORM fails to completely encapsulate database access concerns in object.*** We find that even though ORM tries to abstract database accesses, such abstraction cannot be completely encapsulated in objects. Future studies on the reasons that ORM fails to encapsulate database accesses in objects would help improve the design of ORM frameworks.

- ***ORM cannot completely abstract the underlying database technology.*** Even though ORM ideally should ensure that the code is database-independent (i.e., porting a system to a different database technology should require no code changes), we still find some counterexamples. Future studies may study the reasons that ORM fails to completely abstract the underlying database technology, and help developers create tools to migrate seamlessly to different database technologies.

- ***Although ORM code is frequently modified, there is a lack of tools to help prevent potential problems after ORM code changes.*** In RQ2, we found that ORM code is modified more frequently than regular code, and some types of ORM code are modified even more frequently. However, since changes to ORM model or query code may introduce runtime exceptions that affect the quality of a system, ORM code would benefit from type checking at compile time. Future research on providing automated tools to detect such problems can greatly reduce ORM maintenance effort, and improve the quality of systems that make use of ORM frameworks.

- ***Traditional static code analyzers need to be extended to better capture the peculiarities of ORM code in order to find ORM-related problems.*** ORM-related problems may be different, either syntactically or semantically, from the problems one may see in regular code (due to ORM's database abstraction). Thus,

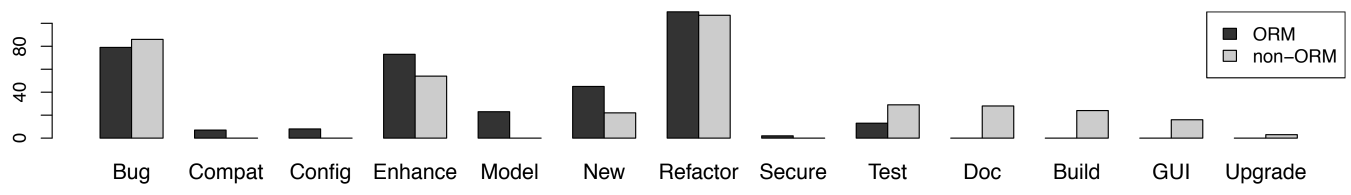


Figure 4: Distributions of the commits for ORM and non-ORM code in each categories.

traditional static code analyzers, which usually do not consider the domain knowledge of ORM code, may not able to detect these ORM-related problems without proper extensions. For example, FindBugs[1] is able to detect security problems in JDBC code, but FindBugs cannot detect such problems in ORM code without a proper extension. A recent study [6] shows that there are many database access code related problems that can be detected using static code analysis; however, existing tools and the research community have not put enough efforts on detecting such domain-specific problems (i.e., related to database access). In general, due to the large number of available frameworks, a better option would be for framework developers to provide static code analyzers for the usage of their frameworks. Thus, developers who are using these frameworks can benefit from these code analyzers when developing their own systems.

- ***Developers may benefit from tools that can automatically help them tune ORM performance configuration code.*** We found that developers are more likely to change ORM code for performance reasons compared to regular code. However, in RQ2 we find that developers do not often change ORM configuration code. Prior studies [52, 13] from the database community show that tuning the performance of database-related code (e.g., SQL) is a continuous process, and needs to be done as systems evolve. In addition, in our prior study [7], we find that developers may not always be aware of the performance impact of the ORM code due to the database abstraction. Therefore, there may be many potential places in the code that require performance tuning. Hence, tools that can automatically change/tune ORM configuration code can be beneficial to developers.

Our industry partner is now aware of the high scatteredness of ORM code changes, and such changes are considered much riskier and require careful attention and review. In addition, our industry partner also recognizes the benefit of having tools that can automatically help refactoring and finding problems in ORM code.

6. THREATS TO VALIDITY

We now discuss the threats to validity of our study.

Internal Validity. In this paper, we study the characteristics and maintenance of ORM code. We discover that ORM code exhibits different patterns compared to non-ORM code. However, we do not claim a casual relationship. There may be other confounding factors that influence our results (e.g., developers intentionally allocate more resources to the maintenance of files with ORM code). Controlled user studies are needed to examine these confounding factors.

[1]http://findbugs.sourceforge.net/

External Validity. To extend the generalizability of our results, we conduct our study on three open source systems and one large-scale industrial system. We choose these studied open source systems because they either have longer development history or are similar to the industrial system. There may be other similar Java systems that are not included in our study. Hence, in the future, we plan to find and include more systems in our study and see if our findings are generalizable. However, our current findings are already having an impact on how our industrial partner maintain ORM code.

We focus our study on JPA (Java ORM standard) because it is widely used in industry and is used by our industrial partner. However, our findings may be different for other ORM technologies (e.g, ActiveRecord or Django). Nevertheless, although the implementation details are different, these ORM frameworks usually share very similar concepts and configuration settings.

Construct Validity. We automatically scan the studied systems and identify ORM code. Therefore, how we identify ORM code may affect the result of our study. Although we use our expert knowledge and references to identify ORM code [43], our approach may not be perfect. We plan to investigate the accuracy of our approach for identifying ORM code changes in the future. We annotate a commit as an ORM-related commit if the commit modifies ORM code. Given the large number of commits (over 10K), we have chosen to use an automated approach for commit classification. It is possible that some modifications in the commit are not related to ORM. However, during our manual study in RQ3, we find that our approach can successfully identify ORM-related commits, and we believe our automated approach has a relatively high accuracy.

We compare ORM code with non-ORM code, but there may be many kinds of non-ORM code (e.g., GUI or network). However, since we are not experts in the studied systems, and there can be hundreds of different sub-components (depending on how we categorize non-ORM code), we choose to categorize code as ORM and non-ORM code. Ideally we would want to compare ORM code with other types of database access code (e.g., JDBC). However, most systems are implemented using only one database access technology, and it is not realistic to compare two different systems that use two database access technologies.

7. RELATED WORK

In this paper, we study the characteristics and maintenance of ORM code, which is not yet studied nor well understood by researchers and practitioners. In this section, we survey prior studies on the evolution of database code and non-code artifacts. While many prior studies examined the evolution of source code, (e.g., [18, 20, 37]), this paper studies the evolution of software systems from the perspective of the non-code artifacts. Such non-code artifacts are extensively used in practice, yet the relation between such artifacts and their associated source code is not widely studied.

7.1 Evolution of database code

Prior studies focus on the evolution of database schemas, while our paper focuses on the evolution of the database-related code. Qiu *et al.* [45] conduct a large-scale study on the evolution of database schema in database-centric systems. They study the co-evolution between database schema and application code, and they find that database schemas evolve frequently, and cause significant code-level modifications (we observe similar co-evolution even though ORM is supposedly designed to mitigate the need for such co-evolution). Carlo *et al.* [12] study the database schema evolution on Wikipedia, and study the effect of schema evolution on the system front-end. Curino *et al.* [11] design a tool to evaluate the effect of schema changes on SQL query optimization. Meurice and Cleve [40, 19] develop a tool for visualizing database schema evolution. Goeminne *et al.* [21] analyze the co-evolution between code-related and database-related activities in a large open source system, and found that there was a migration from using SQL to ORM. In another work, Goeminne *et al.* [22] study the survival rate of several database frameworks in Java projects (i.e., when would a database framework be removed or replaced by other frameworks), and they found that JPA has a higher survival rate than JDBC.

7.2 Non-code artifacts

User-visible features. Instead of studying the code directly, some studies have picked specific features and followed their implementation throughout the lifetime of the software system. Antón *et al.* [1] study the evolution of telephony software systems by studying the user-visible services and telephony features in the phone books of the city of Atlanta. They find that functional features are introduced in discrete bursts during the evolution. Kothari *et al.* [35] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of these features. Greevy *et al.* [23] use program slicing to study the evolution of features. His *et al.* [28] study the evolution of Microsoft Word by looking at changes to its menu structure. Hou *et al.* [29] study the evolution of UI features in the Eclipse IDE.

Communicated information. Shang *et al.* [47, 48] study the evolution of communicated information (*CI*) (e.g., log lines). They find that *CI* increases by 1.5-2.8 times as the system evolves. Code comments, which are a valuable instrument to communicate the intent of the code to programmers and maintainers, are another source of *CI*. Jiang *et al.* [32] study the evolution of source code comments and discover that the percentage of functions with header and non-header comments remains consistent throughout the evolution. Fluri *et al.* [14, 15] study the evolution of code comments in eight software systems. Hassan *et al.* [25] propose an approach to recover co-change information from source control repositories, and Malik *et al.* [39] study the rational for updating comments. Ibrahim *et al.* [31] study the relationship between comments and bugs in software systems.

8. CONCLUSION

Object-Relational Mapping (ORM) provides a conceptual abstraction between database and source code. Using ORM, developers do not need to worry about how objects in Object-Oriented languages should be translated to database records. However, when cooperating with one of our industrial partners, we observed several difficulties in maintaining ORM code. To verify our observations, we conducted studies on three open source Java systems, and we found that the challenges of maintaining JPA code (i.e., ORM standard APIs for Java) is a wide ranging concern. Thus, understanding how ORM code is maintained is important, and may help developers reduce the maintenance costs of ORM code. We found that 1) ORM code changes are more scattered and complex in nature, which implies that ORM cannot completely encapsulate database accesses in objects; future studies should study the root causes to help better design ORM code, especially in Java systems; 2) even though ORM ideally should ensure that the code is database-independent, we find that it is not always true; 3) ORM query code is often changed, which may increase potential maintenance problems due to lack of return type checking at compilation time; 4) traditional static code analyzers need to be extended to better capture the peculiarities of ORM code in order to find ORM-related problems; and 5) tools for automated ORM performance configuration tuning can be beneficial to developers. In short, our findings highlight the need for more in-depth research in the software maintenance communities about ORM frameworks (especially given the growth in ORM usage in software systems).

Acknowledgments

We are grateful to BlackBerry for providing data that made this study possible. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of BlackBerry. Moreover, our results do not in any way reflect the quality of BlackBerry software products.

9. REFERENCES

[1] A. Anton and C. Potts. Functional paleontology: the evolution of user-visible system services. *IEEE Trans. on Softw Eng.*, 2003.

[2] C. Bauer and G. King. *Hibernate in Action*. In Action. Manning, 2005.

[3] S. Boslaugh and P. Watters. *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[4] T.-H. Chen, N. Meiyappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proc. MSR 2014*.

[5] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora. CacheOptimizer: Helping developers configure caching frameworks for Hibernate-based database-centric web applications. `http://petertsehsun.github.io/papers/cacheOptimizer.pdf`. Technical Report.

[6] T.-H. Chen, S. Weiyi, A. E. Hassan, M. Nasser, and P. Flora. Detecting problems in database access code

of large scale systems - an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[7] T.-H. Chen, S. Weiyi, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proc. of ICSE 2014*.

[8] N. Cliff. *Ordinal methods for behavioral data analysis.* Psychology Press, Sept. 1996.

[9] B. Commerce. Broadleaf commerce. `http://www.broadleafcommerce.org/`, 2014.

[10] J. Community. Hibernate. `http://www.hibernate.org/`, 2013.

[11] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *in Proc. VLDB Endow.*, 2008.

[12] C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo. Schema evolution in Wikipedia: toward a web information system benchmark. In *Proc. ICEIS 2008*.

[13] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, VLDB '04, pages 1098–1109. VLDB Endowment, 2004.

[14] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proc. WCRE 2007*.

[15] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Control*, 2009.

[16] A. S. Foundation. Apache OpenJPA. `http://openjpa.apache.org/`, 2013.

[17] E. Foundation. Eclipselink. `http://www.eclipse.org/eclipselink/`, 2014.

[18] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proc. ICSM 1997*.

[19] M. Gobert, J. Maes, A. Cleve, and J. Weber. Understanding schema evolution as a basis for database reengineering. In *Proc. ICSM 2013*.

[20] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of ICSM 2010*.

[21] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *Proceedings of the Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, pages 353–357, Feb 2014.

[22] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in java projects. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 551–555, Sept 2015.

[23] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing software evolution through feature views: Research Articles. *J. Softw. Maint. Evol.*, 2006.

[24] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of ICSE 2009*.

[25] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11(3):335–367, Sept. 2006.

[26] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, pages 510–518, Sept. 1981.

[27] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.*, 2013.

[28] I. His and C. Potts. Studying the evolution and enhancement of software features. In *Proc. ICSM 2000*.

[29] D. Hou and Y. Wang. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *Proc. CASCON 2009*.

[30] IBM. Websphere. `http://www-01.ibm.com/software/ca/en/websphere/`, 2014.

[31] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. On the relationship between comment update practices and software bugs. *J. of Systems and Softw.*, 2012.

[32] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In *Proc. of MSR '06*.

[33] R. Johnson. J2EE development frameworks. *Computer*, 38(1):107–110, 2005.

[34] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.*, 2007.

[35] J. Kothari, D. Bespalov, S. Mancoridis, and A. Shokoufandeh. On evaluating the efficiency of software feature development using algebraic manifolds. In *Proc. ICSM 2008*.

[36] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1988.

[37] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Proc. ISSM 1997*.

[38] O. S. Ltd. JPA performance benchmark. `http://www.jpab.org/All/All/All.html`, 2014.

[39] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. Hassan. Understanding the rationale for updating a function's comment. In *Proc. ICSM 2008*.

[40] L. Meurice and A. Cleve. Dahlia: A visual analyzer of database schema evolution. In *Proc. CSMR-WCRE 2014*.

[41] S. Nakagawa and I. C. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 2007.

[42] J. Nijjar and T. Bultan. Data model property inference and repair. In *Proc. ISSTA 2013*.

[43] ObjectDB. JPA2 annotations - the complete reference. `http://www.objectdb.com/api/java/jpa/annotations`, 2014.

[44] D. Portal. Devproof portal. `https://code.google.com/p/devproof/`, 2014.

[45] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proc. ESEC/FSE 2013*.

[46] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.

[47] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proc WCRE 2011.*

[48] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *J. of Softw.: Evolution and Process*, 2014.

[49] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proc. WOSP 2010.*

[50] Z. Soh, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. Towards understanding how developers spend their effort during maintenance activities. In *Proc. WCRE 2013.*

[51] ThinkGem. JEEsite. `http://jeesite.com/`, 2014.

[52] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer. *IEEE Data Engineering Bulletin*, 2008.

[53] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: a case study on firefox. In *Proc. MSR 2011.*

[54] ZeroturnAround. Java tools and technologies landscape for 2015. `http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/`, 2014.