Are the Methods in Your Data Access Objects (DAOs) in the Right Place? A Preliminary Study

Maurício F. Aniche, Gustavo A. Oliva, Marco A. Gerosa
University of São Paulo
Department of Computer Science
E-mails: {aniche, goliva, gerosa}@ime.usp.br

***Abstract*—Isolating code that deals with system infrastructure from code that deals with domain rules is a good practice when developing applications. Code that deals with the database, for example, is often isolated in classes following a Data Access Object (DAO) pattern. Developers often create a DAO for each domain entity in the system. However, as some pieces of code deal with more than one entity/table, developers need to decide in which DAO they will place the code, and sometimes choose a less intuitive location. In this paper, we present a heuristic to identify methods that may have been written in an ambiguous place. To validate the idea, we tested it on three industrial projects from a Brazilian company. The heuristic selected, on average, 13% to 18% of the methods in DAOs. After evaluating such methods, we concluded that the heuristic was correct in 50% to 75% of cases. Therefore, we believe that the heuristic can indicate possible technical debt, where the developers may inspect and possibly refactor.**

I. Introduction

Object-oriented software development taught us about the separation of concerns [4] [1] [8] [3]. Today, it is common sense among developers to separate code that deals with system infrastructure from code that deals with business rules and domain objects. Database access, for example, may require large and complex code and the use of external libraries, so developers usually isolate it in Data Access Object (DAO) classes. Moreover, as the number of database queries in a system can be huge, the data access code is split into many different DAO classes, usually one per table/entity. For example, the *InvoiceDAO* is responsible for persisting and retrieving data from invoices, while the ProductDAO is responsible for dealing with products.

However, a query can be complex and make use of many different tables. When that happens, a question that often comes to developers' minds is **in which DAO should I put this query?** In practice, what happens is that developers sometimes write the method in the wrong -— or in an ambiguous —- DAO.[1] A possible explanation for this would be that, at the time the developer was writing the query, s/he put the method on the DAO related to the first table in the query.

[1]"Wrong" may be not be the best word to describe this action. It is hard to affirm that a method is in the wrong place, as this depends on the context. By wrong, we mean a method that could possibly be in another DAO, in which it would be easier to find– we use the word ambiguous to express this throughout the paper.

When the location of a method is ambiguous, developers spend much time searching for it. If they do not find it, they will duplicate code and write the same query in a new method. Writing the method in the right DAO avoids code duplication and improves separation of concerns, leading to higher maintainability. Zazworka et al. [9] studied technical debts and their relationship to software quality, and found that a modularity violation such this one, in which the method is located in the wrong module, may be highly related to bugs and a higher change likelihood.

A technical debt (TD) [2] [6] reflects inadequate decisions that were made in the system. Seaman's research group [5] categorizes three different ways to tackle the TD: identification, measurement, and monitoring. In this paper, we focus on TD identification by aiming to find DAO methods in the wrong place. In order to do that, we conceived a simple heuristic, based on the method's return type and input parameters. We evaluated it in three web projects that belong to a Brazilian company. We found that the heuristic selected from 13% to 18% of all DAO methods, and it was correct 50% to 75% of the time in all projects. Thus, it indeed points to places that need further inspection.

This paper is structured as follows. In Section II, we briefly explain DAO classes and why developers sometimes write database access methods in the wrong class. In Section III, we present our experiment and describe, in details, our approach to identifying the wrongly-placed methods. In Section IV, we show our findings based on the three evaluated projects, and we discuss their consequences. In Section V, we present threats to the validity of this study. Finally, in Section VI, we present our final thoughts, as well as plans for future work.

II. Data Access Objects (DAOs)

A Data Access Object (DAO) is an abstraction to a data source, such as a database [7]. The goal of this pattern is to separate the business logic from the database persistence, so that both parts can evolve and be changed independently [4].

This pattern is very common in many applications. As each entity may be queried in many different ways, developers tend to write one DAO class per table/entity, to facilitate maintenance. This means that a system with three entities, such as *Invoice, Product, and Order,* will have three tables to store each of them, and three different DAOs (usually named after the entity plus the suffix "DAO").

This naming convention also facilitates maintenance. If a database query that deals with the *Product* table needs to be changed, then it needs to be done in the *ProductDAO* class. If, for any reason, the query is not there, the developer will have to search for it in the other DAOs, or write a new one.

However, deciding in which DAO a specific query should be is not an easy task. A single query may touch different tables (using the JOIN instruction, for example). In Listing 1, we show an example of that. This query joins three tables: *Projects, Commits, and Artifacts.*

Listing 1: An example of a SQL query that deals with three tables

```
SELECT
  p.name as projectName,
  c.id as commitId,
  a.name as artifactName,
  a.path as artifactPath
FROM
  Projects p
JOIN
  Commits c on c.project_id = p.id
JOIN
  Artifacts a on a.commit_id = c.id
WHERE
  p.repository = 'Apache';
```

If one analyzes the query, s/he may notice that it probably belongs to *ArtifactDAO*. The SQL returns many artifacts and their relative commit and project. This method probably returns a list of *Artifact*. On the other hand, there are two other tables in the query. It would not be an exception to find this query in *ProjectDAO*, or even in *CommitDAO*. This is an example of what we call an ambiguous method.

This technical debt may cost the team. Code duplication, for example, is a possible consequence. When developers open a DAO class and do not find the method they want, they may write a new one, resulting in two different methods that do the same thing in the system.

Based on that, it is clear that developers must find a way to detect methods and queries that were written in ambiguous DAOs. However, we do not aim to find a perfect algorithm to find them: as with any metric, we are looking for good indicators. In practice, if a project contains 1000 methods in its DAOs, it is impossible for a human to check each one of them manually; but, if the algorithm filters 50 methods, a human can verify those at an acceptable cost. With that being said, our goal is *to write an algorithm that selects a reasonably small set of classes that need to be manually checked.*

III. Experiment Design

Based on the discussion and motivations given above, this paper aims to answer the following research question:

> ***RQ.*** *How can one automatically identify methods that may have been placed in the wrong or in ambiguous DAOs?*

To find a solution to that question, we conceived a heuristic and implemented it in a tool that outputs the methods that seem to be in the wrong place. As a case study, we conveniently selected three projects from a Brazilian software development company. Maurício, one of the authors of this paper, works for this company. All three are web projects, developed in Java. They all use the same technology and frameworks, namely VRaptor (MVC Framework), Hibernate (Object-Relational Mapping Framework), JSP, and JSTL (view layer). Each deals with a specific domain: *Caelumweb* is an ERP, *Gnarus* is an e-learning system, and *Codesheriff* is a code metric visualization application. Although the projects were developed by different teams, teams typically share ideas and experiences.

After running the tool on the three projects, we invited a developer from each project to manually inspect all methods that were output. The developer from *Caelumweb* had worked on the project for the previous two years; the developers from *Gnarus* and *Codesheriff* had worked on the projects from their beginnings (three and one years, respectively).

The developers were instructed to state whether each listed method was in the right place. There was no specific technique – they all had the list of all methods selected by the algorithm and the full source code of the project. They then navigated through each selected method (and sometimes through other related classes) and made decisions. As discussed before, it is hard to precisely define whether a method is in the right place, thus the decisions were purely based on the feelings and knowledge of the project of the developers.

If a developer thought the method was in a wrong or ambiguous place, we considered that the heuristic had correctly selected that method. If s/he stated otherwise, we considered that the heuristic was wrong. If, for some reason, the developer was in doubt about the method, we considered the heuristic as correct for that method as well (if even the human did not know if the method was in the right place, such a method definitely merits human analysis). In Table I, we summarize the aforementioned decision algorithm.

TABLE I: Decision table

Suggested method	Developer's opinion	Result
X()	Right place	Heuristic Failed
Y()	Wrong place	Heuristic OK
Z()	Not sure	Heuristic OK

In Table II, we describe the size of each project. The number of classes and number of commits represent the size and age of the project. The total number of DAOs and methods reflects the number of queries and methods that needed to be analyzed. This number varied from 50 to 900 Java classes, and from 10 to 80 DAO classes. *Caelumweb*, for example, contained almost 600 methods in DAOs. If a developer were to go method-by-method to check if the methods were in the right place, s/he would expend a lot of effort.

TABLE II: Analyzed Projects

Project	# of Classes	# of Commits	# of DAOs	# of Methods
Gnarus	924	10451	39	233
Caelumweb	1321	12077	81	590
Codesheriff	56	339	10	70

In the following subsections, we detail the designed heuristic to identify potentially misplaced methods. All the gathered data (including the SQL queries used to manipulate it), as well as the specific build of the tool we employed, are available on a website[2].

A. The Heuristic

The heuristic is based on the assumption that *all queries that deal with the X table/entity should be on XDao*. By dealing with the entity, we mean that the method should return the entity or receive it as an input parameter. Also, in practice, we noticed that many methods return a primitive type. That happens when the query returns a single item, such as an integer or a double (by executing a *COUNT* or a *SUM* in the database).

To be considered "correct," a method should follow at least one of the following rules. To simplify the rule, we will assume that *X* is the type that the *XDao* is associated with.

1) The return type of the method is X.
2) The return type of the method is a primitive.
3) The return type of the method is an enum.
4) The return type of the method is a sub-type of X.
5) The return type is a generic type with more than one type.
6) The return type is a generic type of X, such as List <X >.
7) The return type name contains a substring of X.
8) One parameter of the method is an instance of X.
9) One parameter of the method is a generic type of X, such as List <X >.
10) All parameters of the method are primitives.

If a method, for any reason, does not match any criteria, then it is determined that the method may be in the wrong place, and needs to be validated by a human.

B. Implementation

The heuristic is simple in terms of implementation. Essentially, it deals with methods' return types and parameters. We developed a Java parser that navigates through all the classes in a system and returns all methods that do not match the criteria.

The parser was developed in Java and uses of ANTLR to parse the language. Currently, it is a simple command-line tool that prints a list of methods and their respective classes. The source code is freely available[3].

IV. Results and Discussion

In Table III, we show the results of the heuristic for the three projects. The heuristic pointed that 13% to 18% of the methods were potentially misplaced.

As noted before, to validate the effectiveness of our approach, we invited one developer per project to manually inspect the whole output given by our tool. In Table IV, we show the number of methods on which the developers agreed with the heuristic.

TABLE IV: Developers agreement on the heuristic

Project	# of Inspected Methods	# of Agreements	% of Agreement
Caelumweb	79	59	74.68%
Codesheriff	13	8	61.53%
Gnarus	33	16	48.48%

One may note that the agreement ranges from 48% (CodeSheriff) to 75% (Caelumweb). This means that, in the worst case, 1 out of 2 selected methods were considered to be in an ambiguous place. When analyzing the cases in which the heuristic selected a method considered to be in the right place, we found out a few things:

- There are many DAOs whose name does not match the entity's name. Sometimes the DAO's name is a shortcut to the name of the entity. The DAO that deals with the entity *ParcelaDeBoleto* (which, in portuguese, means "bankslip parcel") was named *ParcelaDao*.
- There are two DAOs assigned to the same entity, each one with its own perspective. As an example, the *Course* entity in *Gnarus* contains both the *CourseDAO*, which contains all queries needed by the front-end application, and the *AdminCourseDAO*, which contains all queries needed by the administration part of the application.
- Queries that were grouped by features. In *Gnarus*, we found a DAO that represented all queries needed by a given feature. This feature, in particular, deals with many entities.
- Data Transfer Objects (DTOs) are used frequently. Report queries are good examples of this. The heuristic tries to guess if a class is a DTO, by matching the name of the entity with the name of the DTO. However, this has only worked for a few cases.

In all such cases, although the developer stated that those methods were in the right place, they may still need attention. New developers, or even developers who are not familiar with specific parts of the system, may struggle to interpret these subtle design decisions behind some DAOs.

We conclude that:

> *It seems to be possible to automatically identify methods that may be located in an ambiguous DAO. Our approach, in particular, filters around 13% to 18% of the all methods in DAOs, and it is correct 50% to 75% of the time. As the number of selected methods is small and the assertion rate is high, it may be worthwhile to allocate a developer to manually inspect the methods and move them to a better location, if necessary.*

V. Threats to Validity

As with any initial research, this study contains a few threats to the validity of the results. However, we believe that

[2]http://www.github.com/mauricioaniche/icsm2014-daos. Last access on June 27, 2014.

[3]http://www.github.com/mauricioaniche/calculadora-de-daos. Last accessed on June 26, 2014.

TABLE III: The numbers of the heuristic execution on the projects

Project	# of Methods	# of Right Methods	# of Wrong Methods	% of Wrong Methods
Caelumweb	590	511	79	13.38%
Codesheriff	70	57	13	18.57%
Gnarus	233	200	33	14.16%

these can all be treated in future versions of this work.

- We evaluated only three projects, and they were all from the same company. As companies usually have a standard method of developing applications, the heuristic may not be valid for projects developed by other companies. More projects need to be analyzed.
- We relied on the point of view of a single developer per project. Although these developers were knowledgeable about the projects, a discussion with the whole teams or with the developers that actually implemented the methods could provide greater insight and a more accurate view of the reasons behind the designs. However, as the developers are potential users of those pieces of code, if they considered the methods misplaced, the heuristics actually identified points needing attention.
- Although we validated the methods that were selected by the heuristic (false positives), we did not evaluate those that were in the wrong place but were not identified by the heuristic (false negatives). A more complete manual analysis should be done.

VI. Conclusion and Future Work

Data Access Objects (DAOs) are abstractions to a data source. This is a well-known pattern used in many different applications. However, as argued in this study, developers sometimes misplace their methods. This makes maintenance difficult, as the team cannot easily locate the queries. In the worst cases, they even write duplicate methods.

In this paper, we propose an approach to quickly and automatically identify methods that may be in an ambiguous location. Calculations showed that our heuristic selected from 13% to 18% of all methods in DAOs, and was correct on 50% to 75% of these selections. It seems feasible to allocate a developer to manually investigate each of the filtered methods, and to move them to a better location, if necessary.

As a next step, we want to ask an outside expert to assess the methods and identify which are incorrect. That would allow us to use standard precision/recall metrics. Understanding the costs of this technical debt is an important step, too; if a method is in the wrong place, but the refactoring is more expensive than the cost of the developer searching for the method, keeping the method in the wrong place may be the best option. In addition, we need to further evaluate this model by analyzing other projects belonging to different companies, as well as open-source projects.

Acknowledgments

We would like to thank *Caelum Ensino e Inovação* for allowing us to run the study in its environment, as well as supporting the development of the tool. We would also like to thank NAWEB, NAPSoL-PRP-USP, CNPq, and FAPESP for their support. Gustavo received a grant from CNPq under the program Science Without Borders (250071/2013-4) during the development of this work.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: from metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[3] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third edition, 2004.

[4] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[5] C. Seaman. Carolyn seaman's research group on technical debt, 10 2012.

[6] C. Seaman and Y. Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[7] I. Sun Microsystems. Core j2ee patterns - data access object, 02 2007.

[8] R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education, 2002.

[9] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.