Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

MongoDB Code Smells

Defining, Classifying and Detecting Code Smells for MongoDB Interactions in Java Programs

Bernard, Jehan; Kintziger, Thomas

Award date:
2021

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Apr. 2024

Université de Namur
Faculté d'informatique
Année académique 2020–2021

MongoDB Code Smells: Defining, Classifying and Detecting Code Smells for MongoDB Interactions in Java Programs

Jehan Bernard
Thomas Kintziger

Maître de stage : Michele Lanza, Csaba Nagy

Promoteur : ______________________ (Signature pour approbation du dépôt - REE art. 40)
Anthony Cleve

Co-promoteur : Boris Cherry

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Acknowledgements

We would like to thank our supervisors, Anthony Cleve and Boris Cherry, as well as our internship masters, Michele Lanza and Csaba Nagy for their advice, their support and their sympathy during the writing of our thesis as well as during our internship. They made it a rewarding and inspiring experience by sharing their ideas and their passion for databases.

A great thanks also to the Software Institute at the USI for their welcoming attitude in Lugano, despite the difficult context due to the pandemic.

We also thank Marie Wouters for all the English corrections she provided to our thesis. Thanks to our families for their unconditional support in the realization of our thesis.

Résumé

La recherche et la détection de code smells est un sujet important dans la maintenance et l'évaluation de la qualité des logiciels. En effet, depuis l'introduction du terme par Martin Fowler et Kent Beck en 1999, il a été largement adopté et de nombreuses recherches sur ces code smells ont été menées. Les systèmes de gestion de bases de données orientés NoSQL (Not only SQL) sont apparus il y a une dizaine d'années et commencent maintenant à susciter l'intérêt des études scientifiques. En raison de l'émergence de nouveaux systèmes de gestion de bases de données, de nouveaux types de code smells doivent être étudiés afin d'éviter leur persistance dans ces nouveaux systèmes. Cette thèse a pour but de présenter les techniques que nous avons définies et implémentées pour détecter différents code smells dans les interactions entre un programme Java et une base de données MongoDB. Nous avons d'abord défini un catalogue pour regrouper et classer les code smells que nous avons pu trouver dans la littérature. Ensuite, nous avons développé des méthodes utilisant CodeQL, un outil d'analyse statique de code, pour détecter les instances de certains code smells que nous avions préalablement choisis dans notre catalogue.

Mots-clés— Analyse statique, Code smells, MongoDB, CodeQL, Taxonomie, Détection, Anti-patterns, NoSQL, Java

Abstract

The code smells research and detection is an important topic in software maintenance and quality assessment. Indeed, since the introduction of the term by Martin Fowler and Kent Beck in 1999, it has been widely adopted and a lot of research about code smells has been conducted. NoSQL (Not only SQL) oriented database management systems appeared about ten years ago and now begin to be a subject of interest in scientific studies. Because of the emergence of new database management systems, new types of code smells need to be studied so that their persistence in these new systems can be avoided. This thesis aims at presenting the techniques we have defined and implemented to detect various code smells in the interactions between a Java program and a MongoDB database. We first defined a catalog to group and classify the code smells we could find in the literature. Then, we developed methods using CodeQL, a static code analysis tool, to detect instances of certain code smells that we had chosen in our catalog beforehand.

Keywords— Static code analysis, Code smells, MongoDB, CodeQL, Taxonomy, Detection, Anti-patterns, NoSQL, Java

Contents

1 Introduction & Motivation 1
1.1 Thesis context 1
1.2 Code smells catalog 2
1.3 Static code analysis 2
1.4 Thesis Structure 2

2 State of the art 5
2.1 Code analysis 5
2.1.1 Dynamic analysis method 5
2.1.2 Static analysis methods 5
2.2 Code smells 8
2.2.1 Code smells in object-oriented programming 8
2.2.2 Code smells in SQL database interactions 9
2.3 Code smells and bugs detection in Java code using static analysis 9
2.4 Existing SQL code smell detection tools 9
2.4.1 PL/SQL, COBOL and Visual Basic static analyzer 10
2.4.2 SQLInspect 10
2.4.3 ORM static analyzer 10
2.5 Code smells in NoSQL database interactions 11

3 Background 13
3.1 MongoDB 13
3.1.1 MongoDB Server 13
3.1.2 MongoDB Drivers 14
3.2 CodeQL 15
3.2.1 QL language 16
3.2.2 Example 16

4 Defining code smells 19
4.1 Methodology to define code smells taxonomy 19
4.1.1 Define a list of code smells and anti-patterns 19
4.1.2 Analyzing the detection level and translating anti-patterns to code smells 20
4.1.3 Classify code smells into groups 21
4.1.4 Provide additional detection features and information 21
4.2 Our list of code smells 21
4.2.1 Separating data accessed together 21
4.2.2 Use of relational collections 22
4.2.3 Abusive use of indexes 23
4.2.4 Storage of easily calculated values 24
4.2.5 Too long document keys 24
4.2.6 Storage of empty values 24
4.2.7 Human readable values 25
4.2.8 Too long attribute names 26
4.2.9 Database accesses spread across the system 26

4.2.10 Inconsistent order of attributes inside a collection 26
4.2.11 Relying on transactions 27
4.3 Our MongoDB code smells taxonomy 28
4.3.1 Classify code smells into groups 28
4.3.2 Provide additional detection features and information 29
4.3.3 Complete taxonomy 29

5 A tool-supported approach for code smells detection 31
5.1 Analysis method choice 31
5.2 MongoDB accesses extraction 32
5.3 Code smell detection 32
5.3.1 Code smells choice 32
5.3.2 Test project creation 33
5.3.3 Detection rules implementation 34
5.3.4 Validating the rules on real projects 38

6 Implementation 39
6.1 Static analysis with CodeQL 39
6.1.1 Possible alternatives 39
6.1.2 Tool selection reasons 40
6.1.3 Technical constraints 41
6.2 CodeQL database extractor 42
6.3 MongoDB accesses classes 43
6.3.1 Use of inheritance and polymorphism 43
6.3.2 Extraction of the name of the collection accessed 44
6.3.3 Extraction of the operation type 45
6.3.4 Notes about data-flow analysis in CodeQL 46
6.4 MongoDB code smell detection rule class 46
6.5 Code smell detection 46
6.5.1 Test project creation 46
6.5.2 Detection rules implementation 47
6.5.3 Test of the rules on a real project 48

7 Application of the analysis on open-source projects and results 51
7.1 Context: open-source projects 51
7.1.1 Projects dataset 51
7.1.2 Selection of the projects 51
7.2 Results 52
7.2.1 Projects compilation and database creation 52
7.2.2 Analysis of the usage of MongoDB drivers 53
7.2.3 Code smells detection 55

8 Limitations, improvements and future works 65
8.1 Improvements of the code smell taxonomy 65
8.1.1 Discover more code smells 65
8.1.2 Impact severity metrics 65
8.2 Improvements of the analysis tool 66
8.2.1 All-in-one tool creation and IDE integration 66
8.2.2 Use of schema inference 66
8.2.3 Improvements of the CodeQL database extractor 66
8.2.4 Include project analysis using other ODMs or drivers 67
8.3 Future works 67

8.3.1 Deeper analysis about the usage of NoSQL databases 67
8.3.2 Empirical studies about the impact of code smells 67
8.3.3 Visualization of the code smells 68
8.3.4 Code smells in other types of NoSQL databases 68

9 Conclusion **69**

Appendices **77**

A MongoDB bad smells taxonomy **78**

List of Figures

2.1 Code snippet of a Java factorial method 6
2.2 While expression AST from the code snippet in Figure 2.1 6
2.3 Factorial CFG from Figure 2.1 7

3.1 Simple representation of a user in a MongoDB database (from MongoDB website[1]) 13
3.2 InsertOne example on collection user presented in Figure 3.1 14
3.3 Find example on collection user presented in Figure 3.1 14
3.4 InsertOne example in Java MongoDB Driver 14
3.5 Find example in Java MongoDB Driver 14
3.6 Representation of the CodeQL analysis process 15
3.7 CodeQL query that retrieves all accesses to println methods 16

4.1 Methodology diagram to define a taxonomy of code smells 19
4.2 Example of operation using lookup in Java MongoDB Driver 22
4.3 Example of entity class corresponding to a relational collection in Spring Data MongoDB 22
4.4 Example of query using MongoDB Java Driver that might suggest the use of relational collections 22
4.5 Example of index creation with MongoDB Java Driver 23
4.6 Example of indexed field in Spring Data MongoDB entity class 23
4.7 Example of UUID as id in Spring Data MongoDB entity class 24
4.8 Example of an UUID inserted as id in a new document 25
4.9 Example of null value inserted in database using MongoDB Java Driver . 25
4.10 Example of human readable values inserted in database using MongoDB Java Driver 25
4.11 Example of long field name in Spring Data MongoDB entity class 26
4.12 Example of query in MongoDB Shell 27
4.13 Example of embedded document 27
4.14 Insertion of documents with an inconsistent field order with the Java MongoDB Driver 27
4.15 Example of transaction with MongoDB Java driver 28

5.1 MongoDB code smells detection methodology 31
5.2 Logical schema of the rule to detect accesses to data separated in different collections 34
5.3 Logical schema of the rule to detect long field names 36
5.4 Logical schema of the rule to detect too long document keys 36
5.5 Logical schema of the rule to detect too many indexes usage 37

6.1 MongoDB code smells detection implementation diagram 39
6.2 CodeQL database extractor process schema 42
6.3 Inheritance between MongoDB access classes 43
6.4 Example CodeQL query that detects calls to MongoDB methods and the name of the collection they access 44

6.5 Example CodeQL data-flow configuration to obtain a collection name . . . 45
6.6 Flowchart of the retrieval of collection name from an ODM entity class or annotation 45
6.7 Class diagram for TooLongDocumentKeys detection rule 46
6.8 Database diagram of the test project 47
6.9 CodeQL query to detect the creation of too many indexes 48

7.1 First filtering of the projects 52
7.2 Second filtering of the projects 53
7.3 Import of MongoDB drivers in projects 53
7.4 Import of MongoDB drivers in subprojects 54
7.5 Usage of MongoDB drivers in projects 55
7.6 Usage of MongoDB drivers in subProjects 55
7.7 Instances detected for each code smell 56
7.8 Number of projects containing each code smell 57
7.9 Example of a detected instance of the *separating data accessed together* smell in asanthamax/mongodbuserstore 58
7.10 Analysis output for the smell detected in Figure 7.9 58
7.11 *Too long field name* instances per project 59
7.12 Example of a detected instance of the *too long field name* smell in ChuangShiTeam/NowUI-Cloud 60
7.13 Analysis output for the smell detected in from Figure 7.12 60
7.14 *Too long document keys* instance detected in EUMSSI/EUMSSI-platform . 61
7.15 Analysis output for the smell detected in Figure 7.14 61
7.16 Example of an index creation detected in bwaldvogel/mongo-java-server . 62
7.17 Analysis output for the example from Figure 7.16 62

List of Tables

4.1 Code smells classification 29

5.1 Results of the manual detection in our project list 33

7.1 *Separating data accessed together* instances detected in asanthamax/mongodbuserstore 57

A.1 MongoDB code smells taxonomy 79

Chapter 1

Introduction & Motivation

Given the significant growth of digital technology in recent years, the volume and exchange of data is becoming more complex to manage. Traditional Database Management Systems (DBMS) are starting to show their limitations when it comes to systems working with large volumes of data and in data-intensive applications. Indeed, this is especially due to the restrictive character of relational schemas which prevents a greater dynamicity and scalability of databases [28].

During the early 2010s, NoSQL (for *"Not only SQL"*) DBMS emerged to offer a strong solution to the dynamicity and scalability problems of these relational databases. Indeed, NoSQL databases offer more flexibility in data management thanks to their *"schema-less"* aspect [28, 41]. Nevertheless, this type of DBMS is relatively recent and still poses many challenges, in terms of research and development, that could be further explored [31].

With the emergence of this new database management system, code quality problems may also occur in the interaction between projects and NoSQL databases due to the lack of knowledge of these systems which leads to improper implementation of queries. Besides, these quality problems could affect the ease of maintenance and evolution. The analysis of NoSQL code smells and anti-patterns could highlight these quality issues during system development and thus facilitate the treatment of these code smells as soon as they are detected in the code.

1.1 Thesis context

Our thesis was conducted during our internship at the *Università della Svizzera Italiana* (USI) as part of a larger research project conducted by this university and the *Université de Namur* (UNamur). This project is the INSTINCT project that aims to develop techniques to analyze and visualize interactions between applications and NoSQL databases. This would provide developers and researchers tools to improve the quality of interactions in applications using NoSQL databases.

The core objective of our internship was to develop a tool-supported approach to detect several code smells in the interactions between *Java projects* and *MongoDB databases*. Our project can be divided into two main steps. The first one is to find existing code smells that could hide in Java projects and organize them in a catalog. The next step consists in implementing the detection of the selected code smells from the catalog.

This thesis enable us to describe the various steps which have lead us to establish our code smells catalog and to create our techniques of code smells detection.

1.2 Code smells catalog

As we discussed in the previous section, we needed to start by defining a catalog of code smells about the interactions between a Java project and a MongoDB database. Since there is no exhaustive code smells catalog in Java applications interacting with a MongoDB database, the idea is to create our own list grouping all these code smells.

However, as stated in the article by Mäntylä et al. [34], using only a simple list to present a large number of code smells can be problematic for classifying and understanding them. The most efficient way is thus to classify them afterwards in a common catalog with categories according to the different relations that can exist between these code smells, which is what we will do in this thesis. Indeed, the use of this catalog makes possible to have a better overview of the existing code smells and therefore to facilitate the prioritization of their treatment.

In addition to the above-mentioned information, the article by Marticorena et al. [37] suggests adding metrics and criteria to refine the classification of code smells. Indeed, according to the article, current taxonomies do not use sufficient criteria to facilitate their classification. Moreover, the article proposes different criteria that could be reused in a taxonomy of code smells. Then, it is interesting to include some of these criteria in our taxonomy to get a better overview of the code smells to be processed first.

1.3 Static code analysis

The second step in our process is the creation of methods for detecting code smells that we chose from our previously created taxonomy. To detect these MongoDB code smells in Java projects, it is interesting to use a code analysis technique. We also note that we perform this analysis on open-source projects. For such projects, a production database is usually not available, and the database connection needs additional configuration. Therefore, dynamic code analysis is not relevant in this context. Indeed, according to Ball [4], dynamic code analysis is done at program execution, but it is not always possible in our context, so we use static code analysis. Indeed, static code analysis is an alternative to the dynamic analysis which allow, according to Ayewah et al. [3] and Jovanovic et al. [23], to analyze a source code without executing the program. This fits perfectly to our context.
The idea is therefore to create MongoDB code smells detection methods based on a static code analysis technique. Currently, there are many static code analysis tools for a large number of languages and for various uses. In our thesis, we rely on one of these tools, CodeQL, that provides us a framework to define detection rules instead of creating one from scratch.

1.4 Thesis Structure

The next chapter provides an overview of the state-of-the-art in the field of code analysis and code smells detection. This chapter aims to highlight the techniques used in code smell detection and gives an overview of what a code smell is. The following chapter is the *"Background"* which introduces the technologies we used in our thesis. Chapter 4 explains the methodology we created and followed to develop a taxonomy of code smells. Chapter 5 talks about how we designed our detection tool. The sixth chapter *"Implementation"* elaborates on the implementation details of our tool. The chapter on applications, Chapter 7, gives the results obtained by applying our tool to a large number of real-world Java projects. Chapter 8 discusses the different limitations we had to deal with in the realization of our tool and the possible ways to overcome them. This

chapter also gives different future works that could be realized in the continuity of our work. Finally, Chapter 9 concludes this thesis.

Chapter 2

State of the art

The purpose of this chapter is to highlight and introduce the different research works that have been conducted in the field of code smell analysis. We start by presenting the existing code analysis techniques and then we present what a code smell is based on the existing literature. Then, we present static code analysis tools in Java and code smell detection tools in SQL. We conclude by showing the gap that exists regarding the research on code smells in NoSQL.

2.1 Code analysis

This section aims to explain the different techniques that could be used to analyze a program source code. We discuss their advantages, limitations, and how they work.

2.1.1 Dynamic analysis method

According to the article [4] by Ball, the definition of a dynamic analysis method is the analysis of a program during its execution by deriving properties that hold for one or more executions. The advantage of this technique is that it is looking at specific execution scenarios or aspects by instrumenting the program, which provides a good precision of the information.

Nevertheless, this technique has limitations. First, it has worse path coverage than static code analysis (see Section 2.1.2) because, by definition, dynamic analysis focuses on specific execution scenarios and only goes through the path defined by this scenario [4]. Also, this technique is quite time-consuming due to the large number of required test cases and due to the execution time [70]. Another limitation is the fact that dynamic analysis needs to execute the program and thus sometimes needs to have access to the potential databases it uses, which are not always available.

2.1.2 Static analysis methods

Since our work mainly focuses on the static analysis of the project, we define and explain this concept on the basis of the existing literature.

A first interesting step is to define what static analysis is. According to the definitions given by Ayewah et al. [3] and Jovanovic et al. [23], it is a methodology that consists in analyzing abstraction of source code without executing the program and without any particular input in order to detect bad programming practices such as vulnerabilities, errors in this code. Jovanovic et al. [23] argues that the use of automatic static analysis methods comes from the fact that software quality has become an important element in software development. Therefore, manual code review is a rather expensive process.

```java
public static int factorial(int number) {
    int factorial = 1;
    int i = 1;
    while(i <= number){
        factorial = factorial * i;
        i = i + 1;
    }
    return(factorial);
}

```

Figure 2.1: Code snippet of a Java factorial method

Also, as computer systems get larger and more complex, weaknesses in these applications due to poor programming practices can lead to maintainability and security problems, which can become very expansive if not detected in time [23, 32].

How static code analysis works

Static code analysis enables error detection in a program through abstract graphs which either represent source code information or the compiled byte code [32].

As described in the article by Novikov et al. [47], static code analysis needs different types of information to find error patterns. Those are syntactic information, semantic information, and a set of diagnostic rules.

Syntactic information: Syntactic information is mandatory in the static analysis because it makes possible to represent the grammar of the language. Before each analysis, an abstract representation of the source, or of the compiled code is produced in an AST (Abstract Syntax Tree) which will be used in the analysis [47].

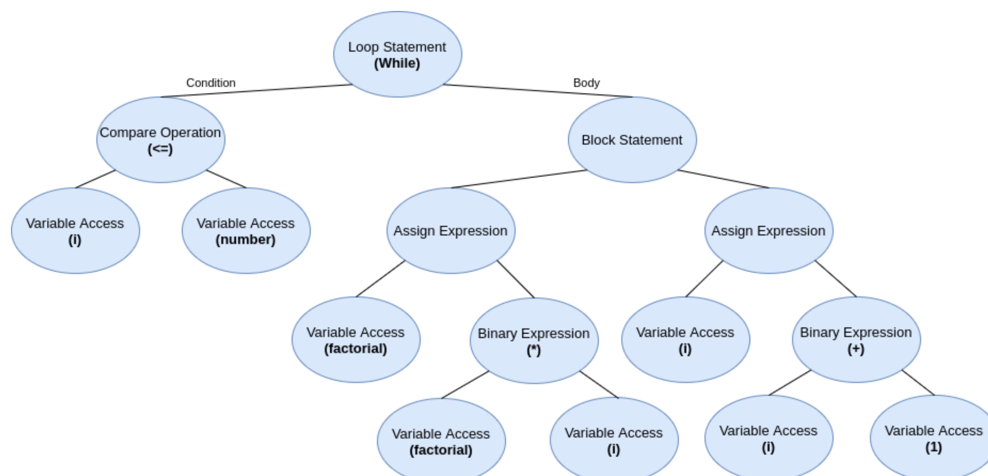

Figure 2.2: While expression AST from the code snippet in Figure 2.1

As we can see in Figure 2.2 the tree has nodes and leaves. This Figure represents the AST derived from the code snippet presented in Figure 2.1. The nodes in AST represent operators, statements, among others and the leaves represent variables and constants.

Semantic information: Semantic information is not mandatory in static analysis but when combined with syntactic information it allows to obtain deeper analyses. This

allows, for instance, to obtain information such as the type of an expression, the signature of a method, the location of the declaration of a variable, or a method [47].

Diagnostic rules: Diagnostic rules are queries used in static analysis to detect errors or patterns in code fragments according to different degrees of problems severity and type [47].

Static analysis methods

There are many static code analysis methods but we will focus on the ones which seem most relevant in the context of our thesis. These are the *Control-flow analysis* and the *Data-flow analysis* [70].

Let us start with the **Control-flow analysis** the most used techniques in static analysis. According to Zhioua et al. [70], the objective of this technique is to know how the procedures in a program call each other and which ones are really called. This technique allows to represent the flow of the program based on the source code or the disassembled binary code in a directed graph: the CFG (Control-Flow Graph) [38, 70].

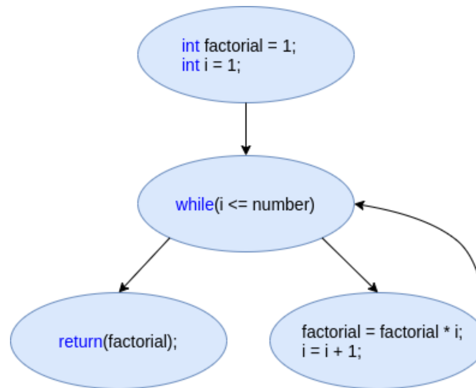

Figure 2.3: Factorial CFG from Figure 2.1

As represented on Figure 2.3, the graph is composed by nodes and edges where a node represents a set of instructions or block of code, and an edge represents a conditional instruction, loops, or branches [38, 70].

The next technique is the **Data-flow analysis** that aims to infer the possible values of a variable at a given moment of the program execution [32, 70]. To do this, it relies on an abstract representation of the program semantic to be analyzed by using a CFG. It will thus represent the dependencies of the data in the source code and will track the effect of the input data, which allows gathering information about the possible values of this data [23, 70].

Strengths

- **Full path coverage**: Static analysis allows to get all the possible interactions between modules or components to find rare pattern occurrences [32].
- **Automated processing**: Automated Static Analysis (ASA) can be developed to automatically detect several types of software anomalies [69, 70].
- **Fast**: By definition, there is no execution time in static code analysis [38]. Moreover, there is the possibility to chose the effective nodes on which the diagnostic

rules will be executed. Thanks to that, this limits the movement in the graph, the access to unnecessary information, etc [47].

Limitation

- **Undecidable and low precision** [32] : it is due to the fact that there is no execution so it is impossible to know every combination of execution scenarios that could occur [38]. In fact, based on Louridas [32], static code analysis is performed with an approximate model representing the program to analyze. Henceforth, because of these approximations, the analysis may miss some cases – which are called false negatives – and may report correct code fragments – which are called false positives.

2.2 Code smells

Since this work aims at detecting code smells in NoSQL database accesses, it is important to specify what exactly is a bad smell and why they are interesting to analyze. The term *code smell* emerged and was popularized by Martin Fowler and Kent Beck [16] in 1999. In their book on code refactoring, they define bad smells as “structures in the code that suggest the possibility of refactoring" before suggesting a list of some common bad smells they have discovered. A bad smell is a fragment of code that may be a source of failures or performance issues in the program and that might be corrected with code refactoring. Bad smells have since been studied for various languages and in different contexts. They are often studied in order to measure maintainability and evolvability [26, 48] or performance issues [20].

2.2.1 Code smells in object-oriented programming

A first taxonomy of code smells in object-oriented programming has been proposed by Mäntylä et al. [34] to bring together the smells presented by Fowler [16] into classes by establishing similar characteristics between them. This taxonomy has been widely used since then and has been refined by the author himself [35]. This taxonomy has also been extended by others such as Marticorena et al. [37], who added several metrics for a better understanding of the smells and of how to detect them. Another taxonomy has later been proposed by Rasool and Arshad [54] based on more abstract concepts which are shared by smells, classifying them into five groups.

The correlation between code smells in object-oriented programming and maintainability of the system has been highlighted in several studies such as the one from Khomh et al. [26] or Olbrich et al. [48] who established a significant correlation between the presence of code smells in a class and the change-proneness of this class. However, change-prone classes tend to produce more defects, and accumulate more technical debt [1]. Empirical studies from Yamashita et al. [65–67] have also demonstrated in specific cases that code smells density of a system can be a good indicator of its maintainability.

Another use of code smells is bug prediction. The influence of code smells on bug proneness of a system has been empirically demonstrated by Khomh et al. [27] and Palomba et al. [50]. A first model of bug prediction based on smells has also been proposed by Taba et al. [61]. As classes containing codes smells are more prone to change, and considering the fact that faults are more likely to occur in change-prone classes [5], we can say that there is a relation between smells and faults.

2.2.2 Code smells in SQL database interactions

Code smells have also been studied in the more specific context of data-intensive systems, and more precisely in the interactions between the system and an SQL database. In 2010, Karwin [24] published a book containing common SQL mistakes and anti-patterns he gathered over time. Those anti-patterns are divided into the categories of logical database design, physical database design, queries, and application development. Another booklet published by Phil Factor [12] in 2014 gathers and describes 150 SQL code smells.

Unlike traditional code smells, SQL code smells are not significantly related to software bugs [45] but have most of the time an impact on the performance of the queries made on the database, and therefore on the system itself. This performance impact has been studied by Lyu et al. [33] and Chen et al. [8]. Another study from Yang et al. [68] analyzed the source of performance bugs in real-world systems and found that more than half of them were caused by misuse of ORM APIs. Another common cause is a bad database or application design. All three of these problems are often highlighted in the program by code smells.

A recent empirical study has been made by Muse et al. [45] about the prevalence, the impact and the evolution of those SQL code smells. They draw the following conclusions: SQL code smells are prevalent in data-intensive software; they have a weak co-occurrence with traditional code smells; they have a weaker association with bugs than traditional code smells; they are more often introduced at the beginning of the software development and they persist longer in the code than traditional smells.

2.3 Code smells and bugs detection in Java code using static analysis

Static analysis has been used for code smells and bugs detection in several tools such as FindBugs, developed by Ayewah et al. [3]. This tool allows programmers to detect more than 400 Java bug patterns in different contexts like bad practice, correctness, internationalization, malicious code vulnerability, multithreaded correctness, performance, security, and dodgy code. It also assigns a severity metric, from low to high, to each bug pattern detected. FindBugs [3] is written in Java and uses various techniques such as the visitor pattern for some simple bug detections or intraprocedural data-flow and control-flow for more complex ones. Since FindBugs is no longer updated, it has been superseded by SpotBugs in 2016.

In the meantime, Fokaefs et al. [14] presented JDeodorant: an Eclipse plugin using the abstract syntax tree of the program to detect the *'Feature envy'* code smell and apply automatic refactoring to correct it. It has been improved over time to include many other smells from Fowler's book such as *'Long methods'* [62], *'God classes'* [15], *'Duplicated code'* [39] and *'Refused bequest'*. [30] We can also cite, among others, PMD [52], inCode or iPlasma [36] that are static detection tools based on software metrics.

Those tools and others have been compared and evaluated in several studies such as Rasool and Arshad [53], Paiva et al. [49] or Hamid et al. [19]

2.4 Existing SQL code smell detection tools

Static code analysis has also been used in some tools for detecting SQL code smells. Therefore, this section presents some of those tools, their purpose and a brief explanation on the way they use static code analysis to find SQL code smells.

2.4.1 PL/SQL, COBOL and Visual Basic static analyzer

A first tool that we could cite is the one created by Van Den Brink et al. [64] in 2007. Its objective is to analyze the quality of the embedded SQL queries inside the systems developed in PL/SQL, COBOL, and Visual Basic. The quality assessment is given by some metrics about quality, reliability, and efficiency. In order to analyze the SQL statement quality, they first extract SQL queries from the source code with control-flow and data-flow analysis. This extraction is made on two types of queries: queries constructed as a string in the host language and queries using a syntactic extension to the host language. They justify this choice by explaining that some programming languages provide a dialect of SQL as a sublanguage to embed SQL queries, while others construct queries as textual strings in the host language. [64]

Once queries have been extracted from the code, they analyze the results to detect bad and good practices in embedded SQL like duplication in the queries, the accesses to different tables, the queries centralization level in the system, the queries number and the number of input variables occurrence. All those elements provide quantitative measures to estimate the quality of the system.[64]

2.4.2 SQLInspect

We can also mention SQLInspect which was developed by Nagy and Cleve [46] in 2018. This tool works as an Eclipse extension that takes advantage of its features to find and highlight common coding mistakes or problematic SQL statements embedded in Java programs (using JDBC, MySQL or Apache Impala) by static query extraction [46]. The creation of this tool is supported by the fact that SQL statements in code are often dynamically constructed by string operations on variables from different locations, so it is complicated to understand the statement that is really sent to the database and to apprehend potential problems that may arise.

SQLInspect works with a static control-flow analysis that extracts SQL queries written in Java. Those queries are, then, analyzed by an internal parser for the supported SQL languages. From the previously extracted queries, the tool can detect SQL bad smells or anti-patterns, as defined by Karwin [24] in his book, and provide SQL quality metrics for those statements. This tool also allows to easily search in the source code for specific queries based on a tree-matching algorithm [46].

2.4.3 ORM static analyzer

The last tool that we cite is a framework by Chen et al. [7]. The framework objective is to detect and prioritize performance anti-patterns inside of systems using ORM[1] for database access by using static code analysis.
The main reason for this framework creation is that many developers use ORMs to ease database access without considering the fact that they can have negative impacts on database performance. Therefore, they track two common types of performance anti-pattern: *Excessive data* – when data is retrieved from a database but never used and *One-by-One Processing* – when operations are performed repeatedly for a single task on similar databases.[7]
This framework uses a data flow and code-path analysis to extract database accesses in the source code. Based on those accesses, the tool uses a set of rules to find ORM performance anti-patterns depending on the type of problem to detect. After that, the

[1]Object-Relational Mapping allows to automatically translate database accesses by mapping application objects to database tables.[7]

tool evaluates the anti-pattern performance impact to provide a prioritization of the bugs that should be fixed [7].

2.5 Code smells in NoSQL database interactions

Despite NoSQL databases becoming more and more common, we found very few studies about the impact of code smells or bad practices on NoSQL databases. The importance of data modeling in such databases has nevertheless been demonstrated by Gómez et al. [17], who compared the performance of six MongoDB databases containing the same data-set but with different data structuring choices. Imam et al. [21] also suggested a list of 23 guidelines to design a document-oriented database. Such guidelines can also be found in the book from Copeland [9] dedicated to design patterns for MongoDB Databases.

If good practices exist for NoSQL databases, we have not found any list of code smells or anti-patterns for such databases. Neither have we found tools aimed at verifying the application of NoSQL good practices in source code.

Chapter 3

Background

The purpose of this chapter is to introduce and explain the main technologies used in our thesis, namely MongoDB and CodeQL.

3.1 MongoDB

As this work only focuses on MongoDB databases, it is necessary to present what MongoDB is, to explain how it works, to give some usage examples and to describe the different MongoDB drivers that exist.

MongoDB is one of the most used databases in the family of NoSQL (“Not only SQL") databases. It is a document-oriented database, which means it stores data in JSON-like documents. Each of these documents represents an object stored in the database and is grouped in collections. Indeed, NoSQL databases do not use tables, columns, rows or schemas for data accesses but use more flexible data models which allow to store a large number of unstructured data [55, 56]. As an example, we can see in Figure 3.1 a simple representation of a user in a MongoDB Database.

```
{
  "_id": "1234",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Figure 3.1: Simple representation of a user in a MongoDB database (from MongoDB website[2])

3.1.1 MongoDB Server

In order to present how MongoDB works, we start with examples of INSERT and FIND methods usages on the user collection illustrated in Figure 3.1. Note that the following

[2] https://www.mongodb.com/what-is-mongodb

queries are presented in the MongoShell[3] language.

```
db.user.insertOne(
    { firstname: "John", lastname: "Doe", address: { street: "123
    Main St", city: "Anytown, USA", state: "Nowhere" ,zip: "000000"
    } }
)
```

Figure 3.2: InsertOne example on collection user presented in Figure 3.1

Figure 3.2 shows an example of *'insertOne'* that inserts a new user "John Doe" in the collection under the same name. As can be seen, we have voluntarily omitted the hobby field to represent the flexibility of NoSQL. In fact, fields in collections do not need to conform to a fixed schema, so fields may differ from one document to another. Therefore, we speak of **schema-less databases**.

```
db.user.find(
    { state: "CA", hobbies: { $in: [ "surfing", "trekking" ] } }
)
```

Figure 3.3: Find example on collection user presented in Figure 3.1

Figure 3.3 illustrates a *'find'* usage whose objective is to obtain all the users from the state of California (CA) whose hobbies are "surfing" or "trekking".

As we can see, the MongoDB queries content is also written in JSON-like format.

3.1.2 MongoDB Drivers

This section is meant to quickly present the Java drivers, which will be analyzed in our work.

Java MongoDB Driver

Java MongoDB Driver is the official driver for MongoDB. This is an API to connect applications and databases by providing access methods on collections. To illustrate how this driver works, we will transpose the selection and insertion methods that we used as examples in Section 3.1.1.

```
collection.insertOne(user)
```

Figure 3.4: InsertOne example in Java MongoDB Driver

```
collection.find(and(eq("state","CA"), in("hobbies", ["surfing", "
    trekking"])))
```

Figure 3.5: Find example in Java MongoDB Driver

As we can see in Figures 3.4 and 3.5, data accesses are made through the "collection" object which represents the user collection. Also, in Figure 3.4, the *'insertOne'* method is called with a Java object "user" in which each field represents a JSON property from this collection.

[3] https://docs.mongodb.com/manual/mongo/

ODM

Like ORM for SQL databases, MongoDB has also a way to automatically translate database accesses by mapping application objects to MongoDB documents. This is called Object Document Mapper (ODM).

In the remaining of our thesis we will analyze three Java frameworks which are based on ODM for database access: *Spring Data MongoDB*, *MongoJack* and *Morphia*.

3.2 CodeQL

CodeQL is a static code analyzer that uses queries to detect potential bugs, errors or vulnerabilities in project source codes. CodeQL, in addition to its standard query library, also offers the ability to create custom queries to identify new kinds of problems. Furthermore, this tool supports various languages like C/C++, C#, Go, Java, JavaScript, Python and TypeScript.[22]

As specified in the CodeQL documentation on GitHub [22], the CodeQL analysis follows multiple steps.

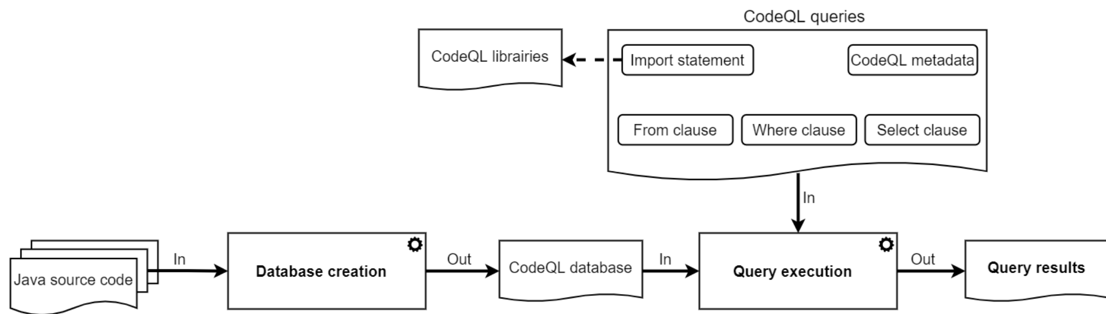

Figure 3.6: Representation of the CodeQL analysis process

Figure 3.6 illustrates the different steps of the CodeQL analysis on Java source code. These steps are represented in bold on the diagram and we describe them in more detail in the following subsections.

Database creation: The first step is the code preparation which consists in the generation of a CodeQL database. For this, CodeQL extracts a relational data model representing each source file in the program source code. The data extracted from the source code is presented in a hierarchical way, including a representation of the abstract syntax tree, the data flow graph and the control flow graph [22].

It should also be noted that for compiled languages, such as Java, the extraction of the database is done directly on the source code. However, the project requires to be build as it relies on the normal project building process to collect each source file and resolve dependencies. This allows a better representation of the program under question [22] .

Query execution: Once the databases are generated, different queries can be executed on them. Those queries are written in a language called QL that we describe in Section 3.2.1 [22]. As shown in Figure 3.6, these queries are made of different elements which are illustrated in Section 3.2.2.

Query results: This step consists in the interpretation of the results obtained after the queries execution, in order to highlight the errors potentially detected by these queries.

Queries contain metadata properties that indicate how the results should be interpreted. Those queries are used to display a message for a single location in the code, or to display several locations that represent steps from a Data-flow or a Control-flow path [22]. The generated results can be represented in output files of different formats such as CSV (Comma-Separated Values) or SARIF (Static Analysis Results Interchange Format) among others [22].

3.2.1 QL language

In order to analyze the complex data structures encoded in the extracted database, CodeQL uses a declarative, object-oriented query language named QL (Query Language). This language has a similar syntax to SQL because they are both query languages. The main difference resides in the semantics of those languages. QL is based on a simple form of logic programming language named "Datalog" which allows to create recursive queries which are called predicates [2, 22, 44]. QL is also an object-oriented language that allows to create reusable queries and to increase modularity without losing its logical aspect. QL allows to define classes that can be inherited: these classes can be considered as predicates and the inheritance can be considered as an implication [22, 44].

3.2.2 Example

As CodeQL is an important part of our work, it seems necessary to present how it works by a simple query example[4].

```
1 /**
2  * @id java/println-method
3  * @kind problem
4  * @description Println method
5  * @problem.severity error
6 */
7 import java
8
9 private class PrintlnMethod extends MethodAccess {
10   PrintlnMethod(){
11       this.getMethod().hasName("println")
12   }
13 }
14
15 predicate isArgumentLiteral(PrintlnMethod method){
16   method.getAnArgument() instanceof StringLiteral
17 }
18
19 from PrintlnMethod printlnMethod
20 where isArgumentLiteral(printlnMethod)
21 select printlnMethod, "Is a println method access"
```

Figure 3.7: CodeQL query that retrieves all accesses to println methods

Figure 3.7 illustrates a CodeQL query written in QL. The objective of this query is to obtain all the accesses to the "println" methods whose arguments are strings.

As we said before, QL is an object-oriented language. Therefore, we can see, on lines 9 to 13, that the access to the "println" methods is represented by a private class

[4]This example has been realized on the LGTM website. The query and its results can be consulted on the site at the following address: https://lgtm.com/query/3380129713944661860/

which extends another class to inherit its properties. In our example, we can see that the “PrintlnMethod" class extends the “MethodAccess" class to specify that the data to be retrieved has to be methods accesses only. Moreover, on lines 10 and 11, the “println" class constructor has been redefined to obtain only the accesses whose method name is “println".

CodeQL also allows the reusability of libraries or modules thanks to import statements, as illustrated in Figure 3.6. In our example, on line 7, we can see that we import the “java" library, which is a standard library provided by CodeQL. In addition, we could also have imported some modules or libraries that we have defined.

We also provided a simple predicate, on line 15, which defines a condition that evaluates whether a method argument is of StringLiteral type. As we said before, predicates represent the logical part of CodeQL.

Finally, CodeQL queries are composed of three important parts: *from clauses*, *where clauses* and *select clauses*.

From clauses: The ‘from’ clause declares the variables in a specified type that will be used in the query [22]. In our example, on line 19, we defined a variable with the type that we created earlier. Thus, in this query, we only use elements of type PrintlnMethod.

Where clauses: The ‘where’ clause is the logical part of the query which generates results that only match the logical conditions defined in this clause. These conditions are applied to the variables defined in the ‘from’ clause [22]. In our example, on line 20, we restrict the results that match with predicates that we have defined before.

Select clauses: The ‘select’ clause is used to display the results that match the logical conditions defined in the ‘where’ clause [22]. In our example, on line 21, we display the object representing the method access and a message. This structure is specified by the @kind property (on line 3) in the metadata. Here, the property states that the ‘select’ clause must always contain an element to display and a string to provide an explanation.

Chapter 4

Defining code smells

The aim of this chapter is to suggest a taxonomy of the MongoDB code smells that we gathered and defined from different sources. To do so, we first present the methodology we followed to find these code smells. Then, we present and explain the different code smells we found. Finally, we explain how we implemented our taxonomy.

4.1 Methodology to define code smells taxonomy

This section explains in detail the methodology we developed and used to create our code smells taxonomy. The process we defined and followed for this purpose is presented in Figure 4.1.

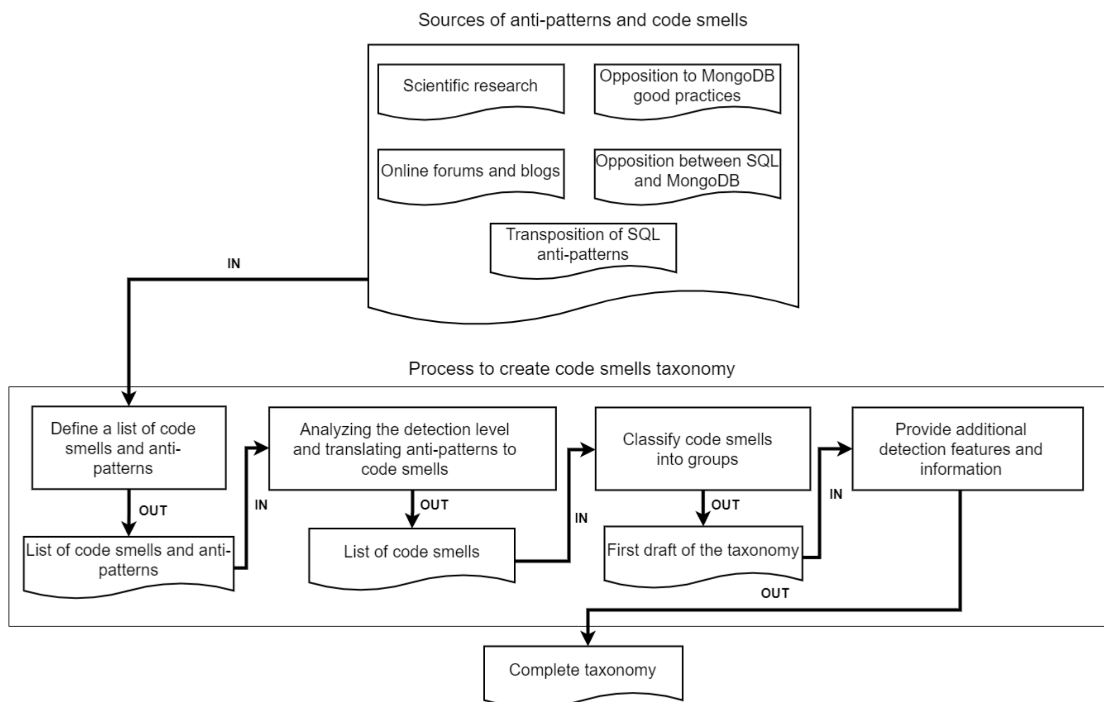

Figure 4.1: Methodology diagram to define a taxonomy of code smells

4.1.1 Define a list of code smells and anti-patterns

To come up with a taxonomy as complete as possible, it is first necessary to define a **list of code smells** based on different types of sources. Therefore, we decided to explore

five types of sources:

Scientific research: Scientific literature is one of the best sources of inspiration for finding information on various scientific topics. Therefore, we first decided to search in the literature to see if some code smells or anti-patterns have already been defined by someone for NoSQL or MongoDB. Despite the many searches we have done, we did not find any.

Online forums and blogs: Online forums and blogs have been a great source of inspiration. Indeed, Mark Kenig, an IBM employee, posted an article [25] on LinkedIn in which he defines a list of Big Data anti-patterns. This post will be our main source for the list of code smells presented in this section. The official website for MongoDB also contains a series of articles dedicated to anti-patterns. They develop six different anti-patterns and explain how to detect them inside a database. Another article posted on InfoQ [13] under the pseudonym of Phil Factor contains a list of some mistakes to avoid while designing or using a MongoDB Database.

Opposition to MongoDB good practices: To complete our list of code smells we also searched for recommended MongoDB good practices. We then investigated whether ignoring those good practices could lead to performance or maintainability issues or not and therefore could be considered as a code smell.

Opposition between SQL and MongoDB: Some other code smells have been defined by searching for errors that programmers used to SQL databases could make when designing or using a NoSQL database the same way they would have done for a SQL database.

Transposition of SQL anti-patterns: Finally, we searched for code smells or anti-patterns already defined for SQL database usages which can be extended to NoSQL databases. This is for example the case for most of the smells based on the architecture of the application using the database rather than on the use of the database itself.

4.1.2 Analyzing the detection level and translating anti-patterns to code smells

The second step, after obtaining a first list of code smells and anti-patterns, is to determine, for each smell, the level at which they can be detected. For this, we have defined three detection levels:

Code Level: The anti-pattern is directly related to the interaction with the database and can be detected inside the source code without inferences about schema or data. Those anti-patterns are the easiest to detect with static analysis.

Schema Level: The anti-pattern is related to the database schema conception. It is then necessary to infer information about the schema to detect those anti-patterns with static analysis.

Data level: The anti-pattern is related to the data that is stored just inside of the database. To detect it, we need information about the database content. For this purpose, we often need either a direct access to the database or a dynamic analysis.

As our objective is to provide a taxonomy of code smells, we converted the anti-patterns from our list to code smells by looking for code fragments that could hint the presence of these anti-patterns. By doing so, we excluded some anti-patterns without a corresponding code smell which, therefore, can not be detected from the source code.

4.1.3 Classify code smells into groups

In order to better classify and understand the code smells that were defined earlier, it is necessary to group them into families within the taxonomy. To do this, it would be interesting to explore two characteristics of code smells: their *origin* and *impact*. Indeed, we noticed that most of the time, code smells with the same origin lead to the same impact. Therefore, a good approach to classify code smells is to group them according to their origin and to give a suitable name to these groups. It allows to have a first draft of the taxonomy that can be completed later.

4.1.4 Provide additional detection features and information

The final step in the development of the taxonomy is to provide additional information about the code smells. Indeed, this information would help to prioritize the processing of code smells to know which ones should be handled before others, and to ease their detection. This information can be about the type of impact of these code smells on the system, the way to detect them, if they violate a basic principle of MongoDB, the degree of difficulty of their detection, or others. All this information can come from personal research or from external sources.

4.2 Our list of code smells

This section contains the list of code smells we have gathered and derived from the sources previously mentioned in Section 4.1.1. For each smell, we provide a short description and an instance of the smell in Java with an explanation.

4.2.1 Separating data accessed together

Description: If normalizing data is the way to go in relational databases, in document-oriented databases like MongoDB, separating data accessed together is most of the time an anti-pattern [58]. Rather than separating data into different collections, MongoDB allows to embed documents to represent one-to-one relationships and to embed arrays of documents for one-to-many relationships. This allows better performances by avoiding using the *$lookup* operator (similar to a left outer join in SQL) which induces slow and resource-intensive queries [58].

In-code symptoms: Accessing data separated in two collections requires the use of the *$lookup* operator. The code on Figure 4.2 contains a simple example of a lookup operation written with the official MongoDB Driver for Java. If we can detect several lookups between the same collections, it means that the data from those collections should probably be stored inside the same collections. An example of query using a lookup operation is provided in Figure 4.2.

Discussion: In some cases, separating data in several collections can be justified in order to avoid embedding huge arrays in documents or exceed the 16MB size limit of MongoDB Document. There also exist design patterns such as the *subset* pattern [10] or the *extended reference* pattern [11] that combines embedding and separating data in

```java
Bson pipeline =
    Aggregates.lookup("joinedColl", "localField", "foreignField",
    "as");
List<Document> articles = collection
    .aggregate(singletonList(pipeline))
    .into(new ArrayList<>());
```

Figure 4.2: Example of operation using lookup in Java MongoDB Driver

several collections to avoid performance and document size issues. The final decision must be taken regarding how the data will be accessed by the program.

4.2.2 Use of relational collections

Description: For the same performance reasons as the previous smell (*Separating data accessed together*), the use of relational collection to represent many-to-many relationships is considered as a bad practice in MongoDB [63]. This data modeling pattern requires a "join" between three collections to retrieve the information about the relation and therefore requires the use of two lookup aggregations.

In-code symptoms: The use of relational collections can be detected in code by looking for a lookup between three collections (see Figure 4.4), or more easily if the program uses an ODM to access the database. It is indeed easy to detect entity classes including only two properties with a name containing "id" or annotated as a reference to another collection. (Figure 4.3)

```java
@Document(collection = "authorsBooks")
public class AuthorsBooks {
    private String _id;
    private String authorId;
    private String bookId;
```

Figure 4.3: Example of entity class corresponding to a relational collection in Spring Data MongoDB

```java
List<Document> authorsAndBooks = collection.aggregate(Arrays.
    asList(new Document("$lookup",
    new Document("from", "authorBook")
            .append("localField", "_id")
            .append("foreignField", "authorId")
            .append("as", "authorBook")),
    new Document("$lookup",
    new Document("from", "books")
            .append("localField", "authorBook.bookId")
            .append("foreignField", "_id")
            .append("as", "books")),
    new Document("$unset", "authorBook")));
```

Figure 4.4: Example of query using MongoDB Java Driver that might suggest the use of relational collections

Discussion: Instead of using a relational collection, we found two possible patterns to implement many-to-many relationships [29]. *Two way embedding* which consists in maintaining a list of references in both collections. For example, in the situation presented in Figure 4.3 we would maintain the list of books written by every author in the author document and in every book document the list of its authors. The second one, called *One way embedding* consists in maintaining such a list in only one of the two collections related – generally in the one "performing" the action associated to the relation. In our example, the authors write the book so the list of books should be embedded in the author document. Once again the decision of embedding only the reference, the complete entity, or just a part of it must depend on the way data will be accessed in the program using the database.

4.2.3 Abusive use of indexes

Description: Indexes in MongoDB are powerful and necessary to improve the performance of queries. Therefore, it is sometimes tempting to create a lot of indexes, even some that are not necessary in prevision of future queries. But creating too many indexes might have the opposite effect and slow down a database. Indexes can take a lot of disk space, especially for the collections having a large number of documents. Indexes also consume RAM as indexes are loaded in memory by the MongoDB engine. Furthermore, as indexes need to be updated by the MongoDB engine, every time the indexed field is changed, they will also consume CPU and increase the duration of field updates. Therefore, it is important to use indexes wisely and only when necessary – for example, on frequently queried data. Creating too many indexes can then be considered as a bad smell [25, 40, 59].

In-code symptoms: Detecting indexes already created in the database might not be possible from the source code but indexes can be created easily inside the program with the "createIndex" method from the MongoDB driver (see Figure 4.5). It is then possible to detect indexes that are created in the code before executing a query and that are not deleted. In projects using ODM mapping, indexes can be defined in the entity classes and then easily detected (see Figure 4.6).

```
1 booksCollection.createIndex(Indexes.ascending("title"));
```

Figure 4.5: Example of index creation with MongoDB Java Driver

```
1 @Document(collection = "books")
2 public class Book {
3     @Id
4     private int _id
5     @Indexed
6     String title;
7     ...
8 }
```

Figure 4.6: Example of indexed field in Spring Data MongoDB entity class

Discussion: It is difficult to determine how many indexes should be considered as too many indexes. It may depend on the number of fields in the documents or on the size of

the collection. The maximal number of indexes that one can create in MongoDB is 64. However, in a video posted by MongoDB employees[1], they recommend not exceeding 50 indexes per collection. To validate this number, it is interesting to evaluate empirically the impact of the number of indexes on a database performance. To avoid impacts from the number of indexes as much as possible and to warn the developer before the situation worsens, we decided to set a threshold of 40 indexes per collection, above which we consider there are potentially too many indexes. As each database is different, the final decision for the appropriate number of indexes for a specific collection still belongs to the person designing it.

4.2.4 Storage of easily calculated values

Description: As in SQL databases, storing easily calculated values is an anti-pattern. Such data will indeed take disk space in the database while they could be easily calculated in the query or in the program.

In-code symptoms: Storage of easily calculated values is more difficult to detect from the code, at least in an automated way because we would have to understand the semantic of the program to know which information is stored in the database and how it is computed. With a manual code review, however, we could detect fields that contains easily calculated values from their name, for example.

4.2.5 Too long document keys

Description: In MongoDB, document keys (ids) are automatically indexed. Therefore, using too long ids will take a lot of space in indexes files in addition to the space they take in each document. A common example is the use of UUIDs. If they are convenient to use, they will rapidly generate large indexes files as the collection will grow. As explained in Section 4.2.3, large index files will induce performance and space consumption problems [25, 63].

In-code symptom: The use of long document keys can be detected by looking for UUIDs. We can look at the declared type for the identifier in entity classes or at the inserted value for the id field of a new document. Figure 4.7 and Figure 4.8 present respectively an example of a Spring Data entity class with an UUID as document key, and the insertion of a new document with an UUID key using the MongoDB Java driver.

```
1 @Document(collection = "books")
2 public class Book {
3     @Id
4     private UUID _id
5     ...
6 }
```

Figure 4.7: Example of UUID as id in Spring Data MongoDB entity class

4.2.6 Storage of empty values

One of the main advantages of MongoDB is schema flexibility [18]. Unlike in SQL databases, the structure of documents inserted in a collection is not fixed, and, therefore,

[1]https://www.youtube.com/watch?v=mHeP5IbozDU

```
booksCollection.insert(
    new Document(_id, new UUID())
        .append(...)
        ...
);
```

Figure 4.8: Example of an UUID inserted as id in a new document

it is not necessary to store empty values if the document does not have an attribute. Not using this possibility may be considered as a code smell as it goes against one fundamental principle of NoSQL databases. Furthermore, storing empty or null values in documents will take unnecessary disk space [25].

In-code symptoms: This anti-pattern is detectable in the insertion of new documents in the database. For instance, if a null value or an empty string is assigned to a field. An example is presented in Figure 4.9.

```
booksCollection.insertOne(
    new Document("_id", 12345)
        .append("title", "NoSQL code smells")
        .append("publication_date", null)
);
```

Figure 4.9: Example of null value inserted in database using MongoDB Java Driver

4.2.7 Human readable values

Description: These code smells consist in storing values in a human-readable format rather than in an optimized format that will take less space in databases. They will take more space, not only in the documents but also in the indexes if those values are indexed. Larger data also induce more network consumption when retrieving it [25].

In-code symptoms: Human readable data is not easy to detect automatically because you have to understand the meaning of the data independently from its type. To detect this code smell, a manual code review would probably be necessary to recognize data stored in a way that could be improved. However, some really basic cases might be possible to discover such as string values like "yes" or "no" used to represent a boolean value, or a sate stored in a string format. The example in Figure 4.10 shows the storage of such values for the *"in_stock"* and *"publication_date"* fields.

```
booksCollection.insertOne(
    new Document("_id", 54321)
        .append("title", "SQL Antipatterns")
        .append("publication_date", "Tuesday, August 03, 2010"
        .append("in_stock", "yes")
);
```

Figure 4.10: Example of human readable values inserted in database using MongoDB Java Driver

4.2.8 Too long attribute names

Description: Due to the use of JSON-based documents to store data in MongoDB, the name of every attribute will be stored for each document. Using long attribute names can then consume a lot of storage [25].

In-code symptoms: The name of the document fields can easily be detected in the program like in the queries accessing the database or inside of entity classes if the program uses an ODM. Depending on the ODM used, the name of the field might be the name of the corresponding entity class attribute or might be defined with an annotation. Figure 4.11 shows an example of a too long field name.

```
1 @Document(collection = "books")
2 public class Book {
3     @Id
4     private int _id
5     ...
6     @Field("international_standard_book_number")
7     private int isbn
8 }
```

Figure 4.11: Example of long field name in Spring Data MongoDB entity class

Discussion: Having meaningful attribute names is important for maintainability and human understanding, but it is important to find a balance between human readability and conciseness. Once again, there is no real consensus about the maximal length a field name should have, and there is no technical limitation for the size of a field name in BSON documents. In order to suggest an ideal length, we examined the document fields used in 250 projects using MongoDB[2]. We compared their name length and choose the 95th percentile as a maximal value. Consequently, we suggest that a field name longer than 20 characters is considered as too long.

4.2.9 Database accesses spread across the system

Description: System maintainability is an important characteristic in software development. Separating access in the database greatly affects the maintainability of the system. For instance, if a change is made to one collection in a NoSQL database, all the accesses to this collection in the system will have to be changed as well. Also, if these accesses are separated in many files, then this can complicate maintenance and impact maintainability.

In-code symptoms: The easiest way to detect the database access distribution in the system is to compare the number of methods that have one or more database accesses to the total number of methods in a system. Another way would be to get the whole database calls locations and compare these locations to ensure that they are grouped in the same place within the file system.

4.2.10 Inconsistent order of attributes inside a collection

Description: Schema flexibility does not mean that there is no need to keep a certain coherence between the documents inside of a collection. It is important to keep the

[2]See Section 7.1.1 for more information about the projects

document fields in the same order because MongoDB accords importance to this order when searching for embedded documents. For instance, the query presented in Figure 4.12 would not match the document presented in Figure 4.13 because the field order is not the same in the query as in the document. Having a different order inside the same collection could then lead to incomplete results.

```
db.books.find(author:{name:"Doe", firstName:"John"}});
```

Figure 4.12: Example of query in MongoDB Shell

```
{
    "_id" : 12345,
    "title" : "NoSQL Code smells",
    "author" : {
      "firstName":"John",
      "name":"Doe"
    }
}
```

Figure 4.13: Example of embedded document

In-code symptoms: This code smell then consist in inserting documents in the same collection but with a different field order. For example, we could see a code like the one presented in Figure 4.14

```
Document author1 = new Document("name", "Bernard")
    .append("firstName","Jehan");
Document author2 = new Document("firstName", "Thomas")
    .append("name","Kintziger");
authors.insertOne(author1);
authors.insertOne(author2);
```

Figure 4.14: Insertion of documents with an inconsistent field order with the Java MongoDB Driver

4.2.11 Relying on transactions

Description: MongoDB has been implementing multi-document transactions since its version 4.0. However, it does not mean that transactions should always be used while interacting with the database. In fact, relying on transactions to ensure data integrity could in some cases be a code smell. Indeed, the need to use transactions frequently may indicate that one of the fundamental rules of MongoDB design – data that is accessed together should be stored together – is not respected [57]. If data accessed together are stored together, there would be no need to use transactions.

In-code symptoms: Figure 4.15 contains an example of transaction use that we found on the MongoDB official website[3]. This code smell can be detected if the same collections are often accessed between the start and the commit of a transaction.

[3] https://www.mongodb.com/transactions

```java
try (ClientSession clientSession = client.startSession()) {
  clientSession.startTransaction();
  collection.insertOne(clientSession, docOne);
  collection.insertOne(clientSession, docTwo);
  clientSession.commitTransaction();
}
```

Figure 4.15: Example of transaction with MongoDB Java driver

Discussion: Like for most of the code smells presented, if the use of transactions can be a code smell, it does not mean it is always bad to use them. The frequency and reason for using a transaction should be considered to determine if there is a conception problem in the database. MongoDB published a whitepaper available online that helps to decide when transactions should be applied or not [43].

4.3 Our MongoDB code smells taxonomy

As stated by Mäntylä et al. [34], code smells presented in a flat list might be difficult to apprehend. To provide a better understanding of the MongoDB code smells defined above, we decided to elaborate a first taxonomy. As explained in Section 4.1, the taxonomy must group smells into categories based on their origin and shared properties. We also provide several features inspired by the taxonomy from Marticorena et al. [37] to improve the understanding of their localization and their impact.

4.3.1 Classify code smells into groups

'Relational design ghosts'

The code smells in this category are: *'Separating data accessed together'*, *'Use of relational collections'*, *'Storage of empty values'* and *'Relying on transactions'*. All those smells share their origin in a misunderstanding of the NoSQL and MongoDB design principles and especially in the use of design patterns that are specific to relational databases. Those smells are common when the programmer is used to designing and query SQL databases and recently switched to MongoDB. All of these code smells, except *Storage of empty values*, violate one of the fundamental rules of MongoDB database design, which is *"data that is accessed together should be stored together"*. As explained before, breaking this rule can lead to performance issues.

Human-oriented decisions

The code smells in this category are: *'Human readable values'*, *'Too long attribute names'* and *'Too long document keys'*. Those three code smells are all data modeling choices that are made to improve the understanding of the database or the ease of use by humans. Those design choices result in data that is not optimized and that may consume more storage, RAM or network bandwidth, and therefore also decrease the performance of the database.

Design oversights

The code smells in this category are: *'Storage of easily calculated values'*, *'Abusive use of indexes'*, *'Databases accesses spread across the system'* and *'Inconsistent order of attributes inside a collection'*. All of those smells result from choices made to accelerate

the development or from skipping the design phase, for the database or the system accessing it. They lead to various problems like a decreased maintainability, bugs, storage waste and performance issues.

Smell type	Code smell
Relational design ghosts	Separating data accessed together
	Use of relational collections
	Storage of empty values
	Relying on transactions
Human oriented decisions	Too long attribute names
	Human readable values
	Too long document keys
Design oversights	Inconsistent order of attributes inside a collection
	Databases accesses spread across the~system
	Storage of easily calculated values
	Abusive use of indexes

Table 4.1: Code smells classification

4.3.2 Provide additional detection features and information

As explained earlier when defining the taxonomy, the objective here is to give additional information on the code smells in order to facilitate the prioritization of their treatment.

Define the impact on the system

For each code smell in the taxonomy, we also assess the kind of impact it might have on the system. We have listed the following kinds of impacts among our code smells: *'Performance'*, *'Maintainability'*, *'Storage waste'* and *'Incomplete results'*.
This information about the code smells impact on the system is derived or is deduced either from what we could find in our sources, or from personal knowledge.

Other features

To complete our taxonomy, we reuse some of the additional features proposed by Marticorena et al. [37] in their taxonomy that we found pertinent to detect MongoDB code smells.

Granularity (Gran.): Granularity, as defined by Marticorena et al. [37], is "the size of the component that suggests bad code smell". As we aim to detect MongoDB code smells in Java programs, we will reuse the same object-oriented levels, which are: system, class and method.

Intra/Inter-relations (Intra.): The Intra/Inter-relation is a Boolean feature that indicates if the code smell can be detected without having information about other related components of the same granularity.

4.3.3 Complete taxonomy

With all the information we have collected from the previous sections, we were able to define a table containing the complete taxonomy and classifying the code smells presented in Section 4.2. This taxonomy is available in the Appendix A.

Chapter 5

A tool-supported approach for code smells detection

This chapter aims to explain the theoretical approach that we intend to follow to implement a tool-supported approach for detecting MongoDB code smells in Java programs.

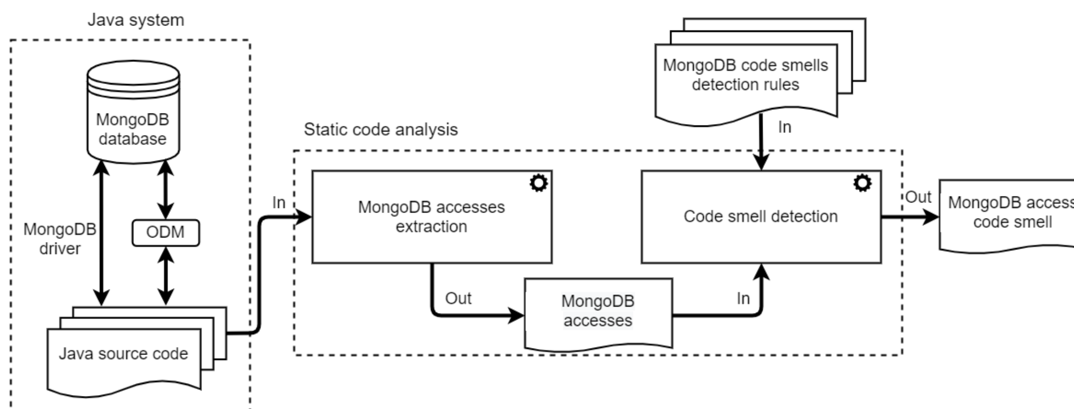

Figure 5.1: MongoDB code smells detection methodology

The schema presented in Figure 5.1 illustrates the methodology of our approach. The realization of this schema was also inspired by the work of Meurice et al. [42] because our detection approach is relatively close to theirs.
As can be seen from the diagram, our detection methodology is divided into several consecutive phases. We discuss each phase in the following sections. We also explain the choice of static analysis for code smells detection.

5.1 Analysis method choice

The code smell detection is based on a static code analysis of Java source code. We preferred this analysis technique to dynamic code analysis for two reasons that we expose in this section.

First, static analysis has the advantage of allowing a quick analyse of the entire source code of a project and an identification of all possible interactions.

Secondly, static analysis does not require the program to be executable and does not require its environment to be configured (e.g. the database connection).

5.2 MongoDB accesses extraction

The first step of our static analysis is to extract from the source-code artifacts where the Java program interacts with the MongoDB database. Since we aim at detecting interactions made with the official MongoDB driver and several ODMs, these interactions may take several forms.

MongoDB Driver and ODM methods: The first code artifacts we want to extract from the source code is the call to methods from the ODMs or the MongoDB driver API, and especially the calls that interact with collections. We can quote, for example, CRUD operations such as *insertOne*, *findOne*, *updateOne* or *deleteOne* from which we extract the collection, the fields and in some cases the type of data. We also need to extract methods used to build queries and methods modifying the collection schema such as index creations.

ODM entity classes declarations and annotations: ODM entity classes can provide a lot of information about the collection schema like the name of the collection, the number of fields, their names and types. It is then important for us to extract them as this information will also be useful for code smells detection. All of the ODMs we want to include in our analysis also use annotations, which makes possible to specify additional information about the database, for example, the fields of the collection that are indexed or if the name of the collection and fields differ from the ones used in the entity class.

String queries: The last artifact we detect are string queries encoded in the source code. In addition to the access methods provided by ODMs and the MongoDB Java Driver, it is possible to execute string queries directly on the database by using BSON (the JSON-based language used by MongoDB). Like for the query methods from ODMs and the MongoDB driver, we extract the collection, the fields and the type of data stored in the fields.

5.3 Code smell detection

The next step of our static analysis is to detect code smells based on the MongoDB accesses – obtained from the previous phase – and on the MongoDB code smell detection rules. To create the detection rules, we took inspiration from the methodology proposed in the article of Novikov et al. [47], who describe an approach to develop these rules in several steps which we will discuss in detail in this section.

5.3.1 Code smells choice

We only had a limited time to realize our work, so we had to choose which code smells we would first detect from the previously defined taxonomy (see Section 4.2). To facilitate our choice, we have defined a set of inclusion and exclusion criteria for the code smells selection below.

Inclusion criteria:

- **One smell per type**: This ensures that we have an example for each smell type.
- **Positive result**: It aims at smells which are present in at least one project. To make sure that these smells are present in the projects, we made a preliminary

manual inspection, and searched for signs or traces of each smell in various projects and we placed our results in Table 5.1.

- **Violating Basics MongoDB Principles (VBMP)**: It ensures that the code smell violates one of MongoDB's core principles. Indeed, we want to detect smells by degree of importance.

Exclusion criteria

- **Implementation Obstacles (IO)**: This is the case if the detection rule – which is designed to find the code smell – requires to understand the meaning of the data that is stored inside of the database. Indeed, such a comprehension is too complex to implement.

- **Non Reproducible (NR)**: This is the case if it is difficult to find occurrences of this smell in the code and if it would be difficult to reproduce this smell in unit tests.

Smell type	Code smell	Detected manually
Relational design ghosts	Separating data accessed together	Yes
	Use of relational collections	No
	Storage of empty values	Yes
	Relying on transactions	No
Human oriented decisions	Too long attribute names	Yes
	Human readable values	No
	Too long document keys	Yes
Design oversights	Inconsistent order of attributes inside a collection	No
	Databases accesses spread across the system	No
	Storage of easily calculated values	No
	Abusive use of indexes	No

Table 5.1: Results of the manual detection in our project list

We decided to apply these criteria to the taxonomy previously defined in order to make a relevant choice. For each smell, we marked in the taxonomy in A if it meets the criteria or not. By doing so, we were able to rank the smells based on the number of inclusion and exclusion criteria they meet, without omitting the criterion stipulating that we want at least one smell per type.

With this rank defined, we chose the code smells we want to detect: *Too long attribute names*, *Too long document keys*, *Separating data accessed together*, and *Abusive use of indexes*.

5.3.2 Test project creation

We developed a set of tests to ensure the behavior of detection rules. For this, we created a test project divided into different parts depending on the code smells to test. Each test is designed to produce positive and negative results. The positive results are the code artifacts that should trigger the detection rule. Likewise, the negative results are the ones that should not trigger the detection rule.

In the following section we describe, for each detection rule, all the cases where the code artifact matches a rule and we provide a simplified logical diagram that represents the rule.

5.3.3 Detection rules implementation

Once the tests are created, the next step is to implement the detection rules based on the smells in the test project. In this section, we will theoretically describe how we intend to detect code smells, in our tool, based on the code artifacts we obtained in the data extraction part. To do this, we discuss the different steps to detect code smells based on a logical schema.

Separating data that is accessed together

As already explained in Section 4.2.1, the access of data separated in two different collections requires the use of *$lookup* aggregations to join them. We then decided to search for the use of those lookups to detect the code smell "Separating data that is accessed together". The detection rule designed to detect this code smell is represented by a logical schema in Figure 5.2 and described in the following section.

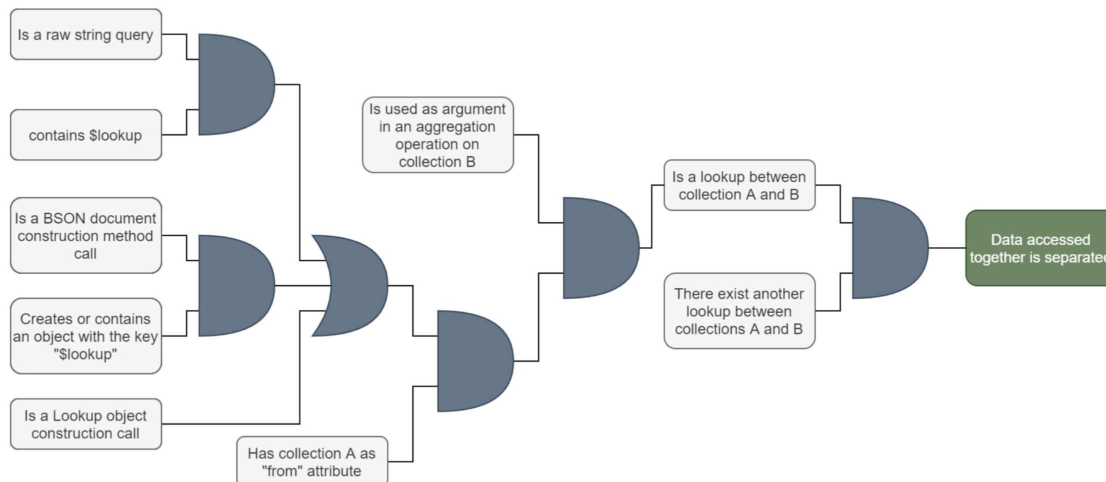

Figure 5.2: Logical schema of the rule to detect accesses to data separated in different collections

First step: detecting a lookup aggregation creation A lookup can be constructed by several code artifacts that we extract from the source code. It can be created, for example, in a string query used as a command on the database. This string query contains a key-value representation of an object in BSON. It will contain two important keywords to detect: "$lookup", which is the key for a lookup object, and "from" which is the key for the collection we want to join, inside the lookup object.

Another way to create a lookup is by using BSON objects constructors. Those constructors take two arguments: the key and the value. Properties can then be added to the BSON object with a method also requiring a key and a value. We can then detect the same "$lookup" key to know that a lookup is created and seek for the value inserted with the key "from" to detect the joined collection.

ODMs and drivers also often provide specific methods to simplify the creation of aggregations. We can then detect calls to the methods, which are specific to lookup creation and obtain the joined collections from the arguments of this method call. For example, Figure 4.2 uses the *Aggregates.lookup* method from the Java MongoDB driver and the collection joined can be extracted from the first argument.

Second step: detecting the collection on which the lookup is executed We then have to detect the collection on which the lookup is executed. For this, we have two possibilities. In some string queries, the collection is specified with the “aggregate" or the “collection" key. The other way is to find with which collection name the collection object on which we execute the query has been instantiated.

Third step: looking for other lookups between the same collections Finally, when we have found the lookup creation in the code and the two collections joined, we check if there exist other lookups between the same collections. If yes, it means that the data from the two collections is accessed together in several ways and that it should probably be stored together. The number of lookups between the two collections can also be used to assess the probability that the two collections should indeed be merged.

Too long field names

The design of the detection rule for too long field names is quite simple and can be explained in two steps: detecting the field names in the source code and verifying their length. Those two steps are explained in more details in this section and are represented by a logical schema in Figure 5.3.

First step: detecting the field names Attribute names can be detected from several code artifacts that we extracted from the source code. The first possibility is to look for BSON objects created in the program and find the keys used for every field. The keys can be detected as explained in Section 5.3.3. However, BSON objects can either represent a document or any MongoDB command or a search filter. To ensure the BSON object we detected is indeed representing a database document, we decided to only keep the objects that are given in arguments to a document insertion method. Field name detection is also possible by analyzing the ODM entity classes. Indeed, an instance of those classes represents database documents. They can provide the field names either from annotations on the class attributes that define the field name in the database, such as @Field or @JsonProperty annotations, or directly from the name of the class attributes, if they are not annotated.

Second step: filtering names that exceed 20 characters in length This step simply consists of verifying for each field names if they exceed the length limit of 20 characters.

Long ID types

To detect the use of large data types as identifiers, we decided to focus on the use of UUID. They are quite popular because they are simple to create and to manage for the programmer as they do not have to search for a meaningful and unique value for their identifiers. They should therefore be the most common large type used as an identifier. Figure 5.4 present the logical schema to detect the use of UUID as document identifier. We can also describe the rule with the following steps:

First step: Detection of the document identifiers Our detection of document identifiers relies on three code artifacts that we will extract. The first artifact covers the BSON creation methods, in which we detect the fields that are inserted with a key ending by “id". The second possibility is to check the ODM entity class fields that are annotated with annotations that declare the document identifier (@Id or @ObjectID) or

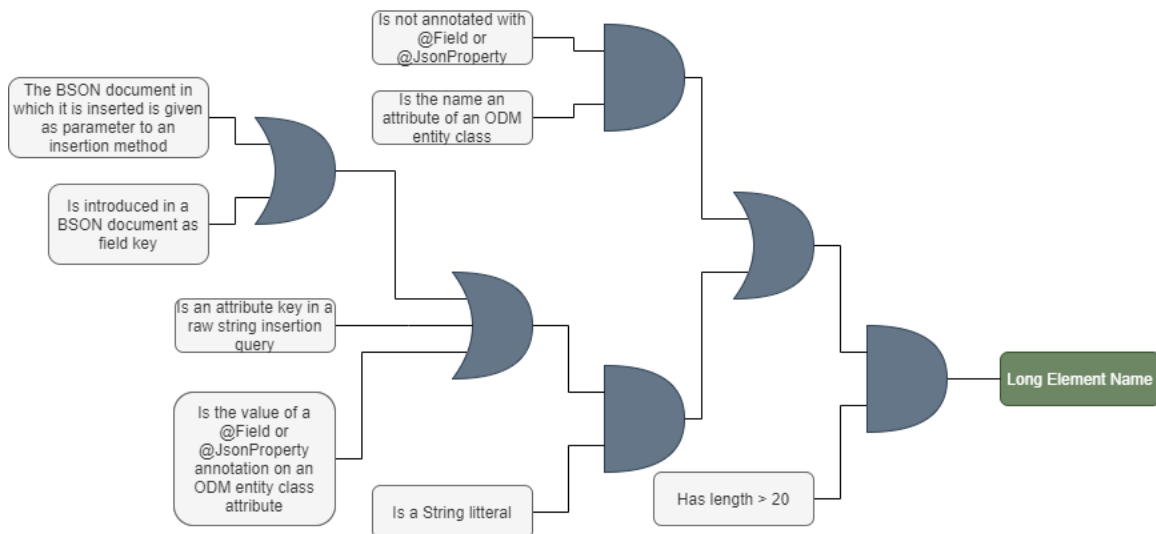

Figure 5.3: Logical schema of the rule to detect long field names

the ones that have a name that ends by "id". The third possibility is to use the MongoDB access methods, by looking for the arguments of methods such as "findById".

Second step: Verification of the identifier type Once we have found the document identifiers, we need to verify if they are UUIDs. For this, we detect two possible cases. The first one is the use of the *java.util.UUID* class. The second case is the use of string identifiers that are assigned in the program with a string representation of a UUID. If the detected artifact corresponds to an identifier, as explained in the first step, and matches one of these two conditions, it will be considered as an instance of the "Long ID Type" code smell.

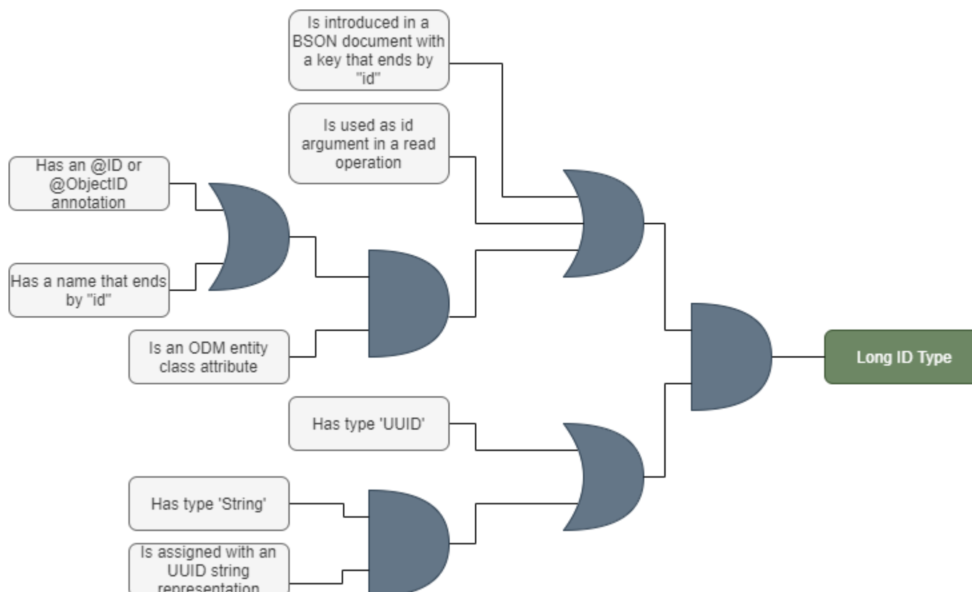

Figure 5.4: Logical schema of the rule to detect too long document keys

Abusive use of indexes

As we said before, the indexes created in the database are not detectable from the code. However, as illustrated on Figure 5.5, the creation of new indexes on a collection can be detected in two ways in the code.

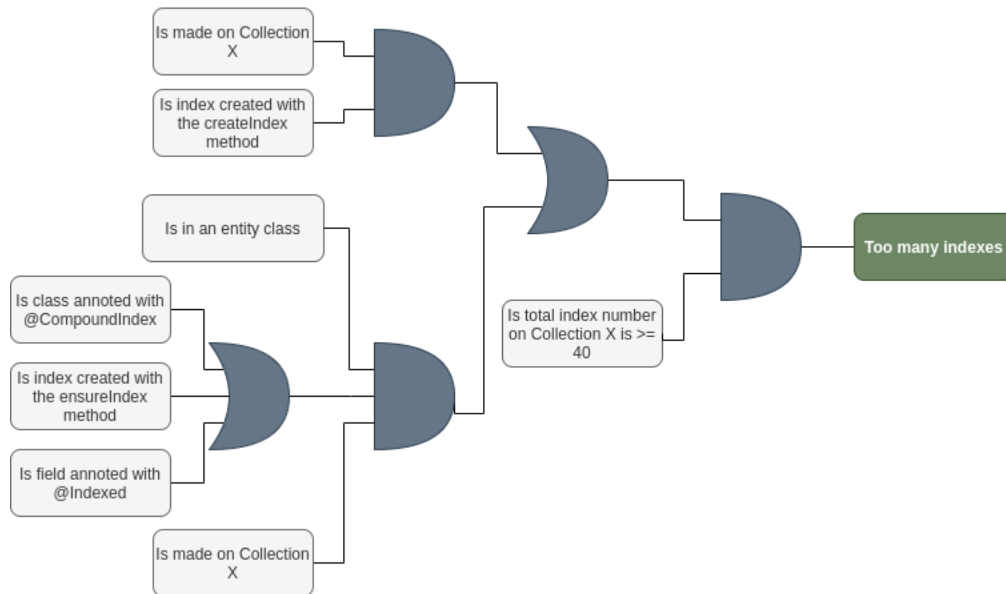

Figure 5.5: Logical schema of the rule to detect too many indexes usage

First step: detection of the index creation The first way, represented on the top of the diagram, is by looking at the use of the index creation methods offered by the MongoDB driver APIs in the code. Indeed, MongoDB Java driver and MongoJack offer methods to easily create indexes on a collection. Those methods are *createIndex* and *createIndexes*. Therefore, the easiest way to detect the index creation would be to find, based on a collection X, all the database accesses that use the index creation methods on this collection.

The other way is by looking at the ODMs, as represented on the bottom of the diagram. In this case, MongoDB collections are represented by application objects – necessarily based on entity classes – on which operations can be performed. Therefore, for Spring Data MongoDB, there are different ways to detect index creation on a collection X :

- by looking at class fields annotated with the annotation *@Indexed*;
- by looking at entity classes annotated with the annotation *@CompoundIndex* which is an index structure composed with a name and all the referenced fields;
- by looking at *ensureIndex* method which allows to create a new index on the collection if it does not exists.

Second step: filtering collections with 40 or more indexes With this set of ways to detect the index creation on collection X, we can calculate its total number of indexes. The code smell should be detected if this number reaches or exceeds 40.

Third step: obtaining additional information about indexes Counting the number of indexes and, when this number exceeds 40, indicating that too many indexes have been created is not always enough. Some collections may have a large number of fields and could, therefore, have a large number of indexes which is not necessarily a design problem. Therefore, it would be interesting to know the exact number of collection fields used to create an index as well as the most used field in these indexes. Thus, one can assess whether the number of indexes reported is problematic or not.

5.3.4 Validating the rules on real projects

To improve the rule quality, we test them on real projects. Indeed, we could discover particular cases that we had not thought of initially and increase the detection of positive results. Therefore, with the knowledge of these new particular cases, it is necessary to readjust the detection rules and the test project.

Chapter 6

Implementation

The purpose of this chapter is to describe how we implemented the approach presented in the previous chapter to detect MongoDB code smells in Java programs. The diagram in Figure 6.1 illustrates the approach we use to detect code smells.

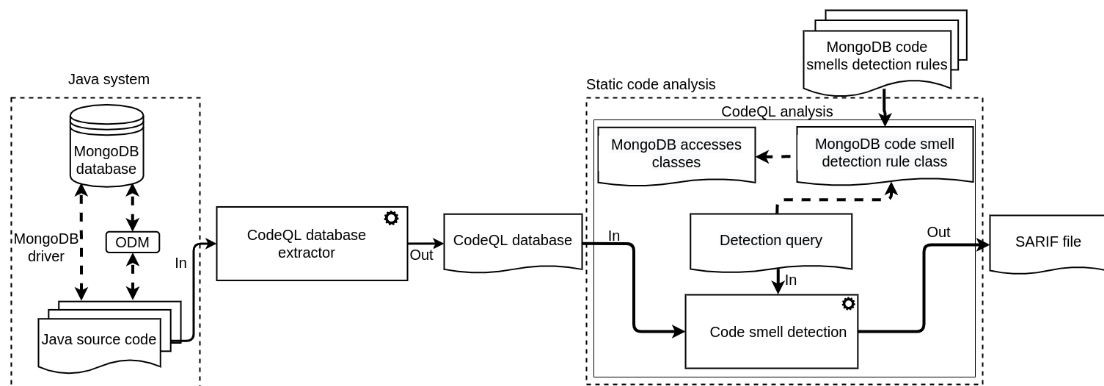

Figure 6.1: MongoDB code smells detection implementation diagram

As can be seen from the diagram, the static analysis method is different from the one theoretically presented in the previous chapter. This is due to the tool we used to perform the static analysis that required us to redesign our approach to fit better with the specificity of this tool. Therefore, we start by presenting the possible alternatives, the reasons why we chose this tool and some technical constraints of it.

After that, we will present each step of our implementation by explaining our choices and illustrating them with various examples.

6.1 Static analysis with CodeQL

Before we start discussing the detection implementation, it is necessary to talk about the tool that we used to develop them. This tool is CodeQL that we have already presented in Section 3.2.

In this section we discuss the different alternatives to CodeQL and the reasons why we chose it rather than the others.

6.1.1 Possible alternatives

Before presenting the advantages and constraints of CodeQL, we highlight the possible alternatives to this static analysis tool. We briefly present such tools by showing their advantages and limitations.

SpotBugs

As we already explained in Section 2.3, SpotBugs is the successor of FindBugs. As explained on the official documentation on GitHub [60], SpotBugs is a static code analysis tool whose objective is to find bugs in Java source code.

Advantages: One of the advantages of SpotBugs is that it is extensible with plugins to create its own detection rules.
SpotBugs fits well to our domain because it also supports the frameworks we intend to analyze like Maven or Gradle [60].

Limitations: A limitation of SpotBugs is that it is restricted to the analysis of source code written in Java. Therefore, it would not be suitable if we wanted to extend our research to other programming languages like Python or JavaScript for instance.

As explained in the SpotBugs documentation, another limitation is the fact that it only fully supports Java versions up to version 11. The most recent ones are still in experimental version [60].

Spoon

According to the article by Pawlak et al. [51], Spoon is a library for transformation and static analysis of Java source code based on an AST analysis. Indeed, Spoon allows to directly edit the source code by manipulating the previously generated AST in order to add various code analyzers and to refactor the code.

Advantages: An advantage of Spoon is the fact that it allows to write its own domain-specific analyses and transformations in a simple way. Moreover, thanks to the technique of transformation and manipulation of the AST, Spoon allows to automatically transform the source to ease the instrumentation or the refactoring [51].

Limitations: Despite the advantages of Spoon, there is still a limitation related to the fact that the only possible analysis is the AST analysis. In our research, it would be necessary to perform control flow or data flow analysis. Indeed, database accesses can be quite dynamic and data values can vary from one execution to another. Our analysis requires to infer how the components act together and to know which values the data could have during the execution for instance.
The other problem, which is the same as SpotBugs, is that the tool only supports the Java language.

Other tools

Considering the large number of static code analysis tools that exist, we cannot cover all possible alternatives to CodeQL. However, an interesting repository on GitHub[1] lists various static analysis tools for different programming languages. Indeed, it could be a good indicator of other possible alternatives to CodeQL.

6.1.2 Tool selection reasons

Although there are alternatives to CodeQL as mentioned above, we decided to work with this tool for various reasons that we will explain here.

[1] `https://github.com/analysis-tools-dev/static-analysis#programming-languages-1`

The first reason why we chose to use CodeQL is that it is a very **powerful and efficient tool** that allows us to easily and rapidly perform different types of analysis on source code such as data-flow analysis, control-flow analysis, AST analysis, or taint tracking. The scalability and the precision of the CodeQL analysis can also be justified by the fact that it is officially used in production by GitHub to detect vulnerabilities.

Another advantage of CodeQL is that it provides many libraries that can be **easily extended** to facilitate query development. Indeed, being able to extend these libraries would allow us to save considerable time in the development of detection rules.

Then, as we specified in Section 3.2, thanks to its object-oriented aspect, CodeQL allows to **reuse components** like custom classes, predicates or queries. Indeed, this specificity allows us to develop unique components that can be easily integrated into other queries, which will facilitate the maintenance and evolution of these queries.

Another important feature of CodeQL is the fact that it **supports various common languages** like Python, Java, C/C++ among others. Indeed, the advantage here is that it would be possible to easily extend our tool to integrate new code smells detection queries for other languages in the future. All of this would be done without changing the static analysis tool for each new language.

CodeQL also provides an **extension for Visual Studio Code** which allows to directly execute queries on the CodeQL database and to directly display the results in a tab. This saves a lot of time when testing queries on the source code.

Finally, the GitHub site offers a **detailed documentation** of CodeQL operation and features to realize custom queries.

6.1.3 Technical constraints

Despite the undeniable advantages of CodeQL, it has some technical constraints that must be taken into consideration for our purposes.

The most important constraint comes from the fact that the static analysis is done on databases generated by CodeQL. Indeed, as a reminder, the analysis performed by CodeQL requires the extraction of relational data from the source code to generate a database.
Moreover, as we also explained in Section 3.2, the database extraction of Java source code requires that it must be compilable in order to generate the databases. However, this compilation can fail for various reasons that must be taken into account in the database extraction.

The first reason is that some open-source project versions are **still under development** and may potentially contain errors. Therefore, due to those errors, it is possible that the compilation of these versions fails and thus does not generate databases.

Another reason is that build systems like Maven or Gradle allow to use dependencie which are external code libraries usable in the code. Therefore, **some dependencies can cause compilation** errors if they are not up to date or if some are not compatible with each other.

Then, Java projects are written and configured in a specific version of this language. Moreover, there are many **Java versions** and many projects developed with one of these versions. Therefore, it is possible that the Java compiler version might not be compatible

with the one used in the project, which will also lead to a compilation error.

The last reason concerns the **Lombok dependency** that is used in some projects. In this case, the projects can be compiled but Java classes that use Lombok features are not taken into account in databases generation by CodeQL. Lombok provides annotations that allow to generate parts of codes without writing them directly thanks to a processor hooked to the Java compiler. However, the analysis performed by CodeQL does not allow to process the Lombok annotations and these classes are simply skipped. This is a known problem that has been addressed in a GitHub issue on the CodeQL repository[2].

6.2 CodeQL database extractor

In this section, we will present the different steps of the extraction process, which we implemented in Python scripts for the thesis. For that, we will base ourselves on the steps illustrated on the diagram in Figure 6.2.

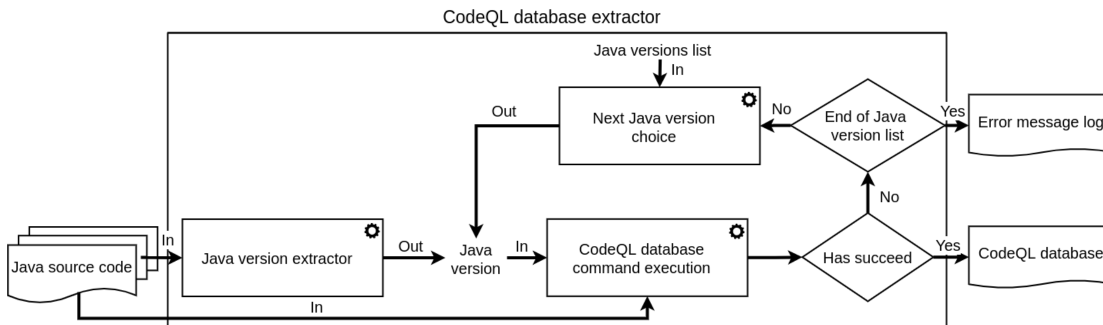

Figure 6.2: CodeQL database extractor process schema

Java version extractor: From the Java source code to be analyzed, the script retrieves the Java version used by the program. To do this, the script first checks if the code is written with Maven or Gradle by looking for a pom.xml or a build.gradle file. Once the configuration file is known, we can find the Java version by observing the tags in these files. The limitation is that developers may use custom tags to define the Java version, which can hinder the detection. For the cases where the Java version cannot be retrieved, we set the output version to Java 8, because it is one of the most frequently used versions.

CodeQL database command execution: The next step of the script is to execute the command to generate the database representing the source code. For that, the script uses the Java version previously obtained and the source code to transform. The executed CodeQL script then automatically detects the project framework in order to execute the right compilation command. If the execution of the command is successful, the database is generated and can be used for the static analysis with CodeQL.

Next Java version choice: If ever the command execution was a failure, all the Java versions between 5 and 16 would be used until we find the one leading to the execution success. However, if all versions have been tested – which means that the entire list has been parsed – and the execution still fails, then the script returns a file containing the

[2]https://github.com/github/codeql/issues/4984

error message log. In that situation, the static analysis is thus not performed for the project.

6.3 MongoDB accesses classes

As explained in our approach (see Section 5.2), the first step of our analysis consists in the extraction of MongoDB accesses by the means of three artifacts in the code:

- MongoDB Driver and ODM methods;
- ODM entity classes declarations and annotations;
- String queries.

To take full advantage of the object-oriented aspects of CodeQL, we created classes that represent each of these code artifacts. By doing this, we extend the CodeQL library, which allows us to query the artifacts represented as easily as any basic program element.

In this section, we present our design and implementation choices for the extraction of MongoDB accesses by using CodeQL classes. We explain the functioning of our classes, justify our choices and provide examples to support our statements.

6.3.1 Use of inheritance and polymorphism

CodeQL being object-oriented means it also provides inheritance and polymorphism. By using these two characteristics, we extended classes from the CodeQL API and created a hierarchy of classes for every artifact we wanted to detect. As illustrated on Figure

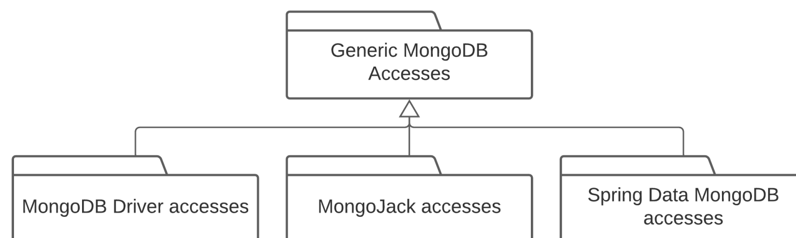

Figure 6.3: Inheritance between MongoDB access classes

6.3, the top classes of our hierarchy are generic classes that match every instance of the artifact, no matter the database driver used. These generic classes define predicates that return information about the artifact that will be necessary for our code smell detection. For example, the generic class that matches MongoDB method calls provides a predicate to obtain the name of the collection accessed by this method call.
However, most of the time the methods defined in the top classes only act as default cases and return basic values, as the way the information needs to be retrieved depending on the driver used, the context and other artifacts characteristics. The objective of our class hierarchy is to match specifically each possible way our artifact can be implemented, and override the predicates declared in the top classes to adapt them to the artifact characteristics. Figure 6.3 represents in an abstract way the structure of our class hierarchy. Inside each driver-specific group, there exists another class hierarchy for more specific characteristics.

Another important mechanism of CodeQL we used at our advantage here is the fact that it determines the most specific class to match when executing a query. A query

```
from MongoMethodCall call
select call, call.getCollectionName()

```

Figure 6.4: Example CodeQL query that detects calls to MongoDB methods and the name of the collection they access

selecting the top class will then automatically return the right child class depending on the context, and therefore execute the right predicate to obtain information. For example, the simple query presented in Figure 6.4 is enough to detect every MongoDB access method call in a program and to determine the collection accessed by each call, regardless of the driver used and how the collection name needs to be retrieved.

6.3.2 Extraction of the name of the collection accessed

Once we are able to detect accesses to the database by the means of drivers in the source code, we still need to know more precisely the collection accessed inside the database to potentially detect the code smells we defined. This can be achieved depending on the type of access.

MongoDB methods on collection objects

In the drivers we analyzed, methods that access a MongoDB collection are called on an instance of a class representing the collection. With the CodeQL library, we are able to access this instance, but the name of the collection can only be found as an argument of the constructor that creates this instance. To obtain this value, we need to do an analysis of the data flow and find string values that flow to the correct argument in the method that obtains or creates our collection object. In CodeQL, this requires to implement a data-flow configuration. Figure 6.5 provides as an example the data-flow configuration explained above, also used to retrieve the name of the collection accessed by a method from the official MongoDB driver.

In this code example, we can see that we need to define a source and a sink to our data-flow path. In our case, respectively a string value and an instance of a collection on which a MongoDB method is called. We also need to define potential additional steps in our data-flow path. For example, in our case, we need to specify that the string value is used as the first argument of the method returning a collection instance.

ODM annotations and classes

When accesses to the database are made through ODM annotations or ODM entity classes, there exist several possibilities to obtain the name of the collection depending on choices made by the programmer. Figure 6.6 represents a flowchart of how the collection name was obtained from an ODM entity class or an ODM annotation. The first step, if we have an annotation, is to find the ODM class in which this annotation is used. Once we have the entity class, we must verify if there is an annotation that defines the name of the collection represented. If not, then it means the ODM automatically uses a collection in the database that has the same name as the class. If yes, we obtain the value specified in the annotation. This value can either be a string literal or a variable or a constant. If it is a string literal, then we have the name of the collection. If it is a variable, we then use the data-flow analysis of CodeQL to trace the value.

```
class MongoDriverCollNameFlowConfig extends DataFlow::
    Configuration {
     MongoDriverCollNameFlowConfig() { this = "
    MongoDriverCollNameFlowConfig" }

     override predicate isSource(DataFlow::Node source) { source.
    asExpr() instanceof StringLiteral }

     override predicate isAdditionalFlowStep(DataFlow::Node node1,
    DataFlow::Node node2) {
         exists(GetCollectionCall call |
           node2.asExpr() = call and
           call.getArgument(0) = node1.asExpr()
         )
       }

     override predicate isSink(DataFlow::Node sink) {
         exists(MongoDriverOperationOnCollectionCall call | sink.
         asExpr() = call.getQualifier())
     }
}

```

Figure 6.5: Example CodeQL data-flow configuration to obtain a collection name

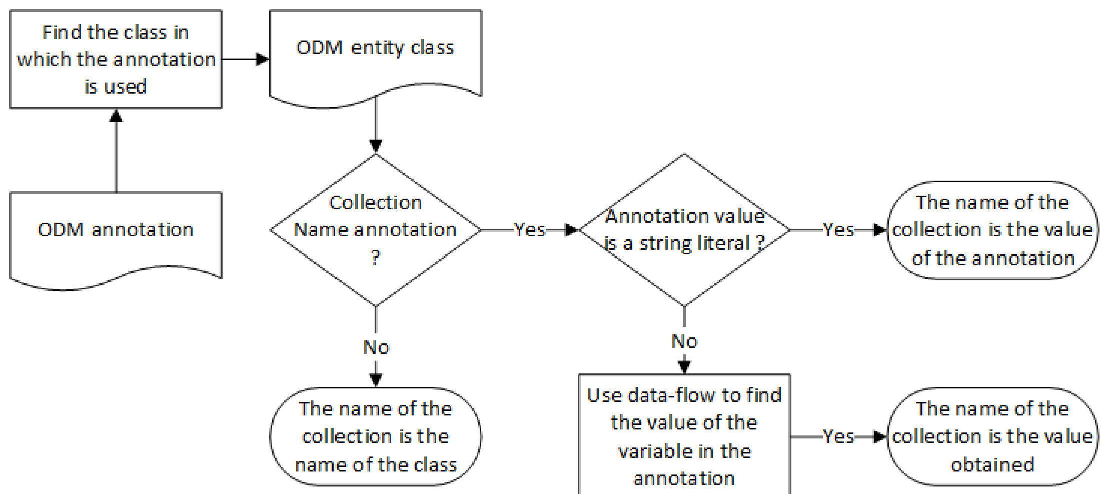

Figure 6.6: Flowchart of the retrieval of collection name from an ODM entity class or annotation

String queries

String queries in the code usually contain the collection on which they are executed. In this case, we use regular expressions to match and extract the value corresponding to the collection name. Some string queries are also executed directly by a method on a collection object. So, we use the method presented above for MongoDB methods.

6.3.3 Extraction of the operation type

Another information we needed to extract about the MongoDB accesses for our code smell detection is their CRUD type. We decided to infer this information from their name

because the methods in the drivers we analyzed follow the same naming conventions. They all start by *"insert"* or *"create"* for creation methods, *"find"* for reading methods, *"update"* or *"replace"* for methods that update documents, and *"delete"* for methods used to delete documents.

6.3.4 Notes about data-flow analysis in CodeQL

While implementing and testing our data-flow analysis, we noticed that CodeQL requires the code to be explicitly called from inside the source code to perform data-flow analysis on it. This can lead to very few results in the case of programs such as APIs that contains methods that are called by the framework only when receiving a request. To improve the recall of our queries, we added multiple ways to detect or infer information when CodeQL was not able to use data-flow. For example, we searched in the AST for initialization or assignations of a variable when CodeQL could not find its value with data-flow analysis. This method being less precise, it is only used in second choice, but from our observations it still provides a decent precision.

6.4 MongoDB code smell detection rule class

Just like the MongoDB database accesses, the code smells detection rules have been represented with CodeQL classes, with the use of inheritance to represent all the possible instances while having one abstract. For each smell, we have an abstract class representing a general instance of the code smell to find and we have classes that inherit from this abstract class representing more specific instances of the code smell.

To illustrate our statements, the example shown in Figure 6.7, represents the hierarchy used for the code smell detection class concerning "TooLongDocumentKeys".

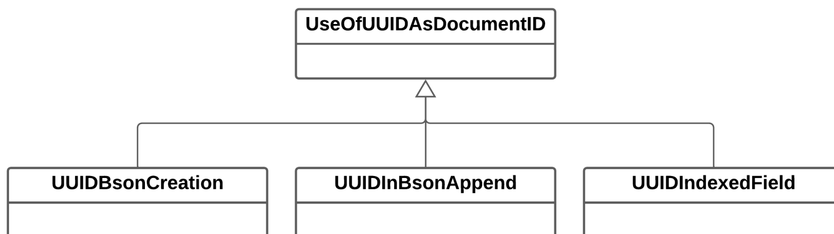

Figure 6.7: Class diagram for TooLongDocumentKeys detection rule

6.5 Code smell detection

The objective of this section is to define and justify the way we have implemented each step, described in Section 5.3. Those steps are: *"Test project creation"*, *"Detection rules implementation"*, and *"Test of the rules on a real project"*.
Indeed, since we have already selected which code smells we wanted to detect in Section 5.3, we omit the step concerning the choice of code smells in this section.

6.5.1 Test project creation

As a reminder, the first step in the development of detection queries was to create a test project. For each test, there is at least one positive and one negative result. To do this,

we created a test project in Java and Maven.

The test project interacts with a MongoDB database, illustrated on Figure 6.8, using the drivers or ODMs, namely MongoDB, MongoJack and Spring Data. For each, we have created a class representing a collection of the database. Hence we can test different cases for all the drivers or ODMs.

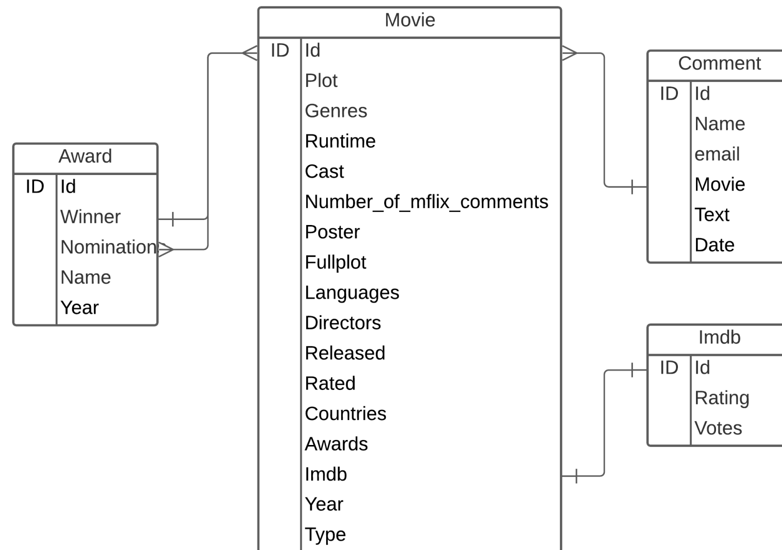

Figure 6.8: Database diagram of the test project

Finally, we created a package for each code smell, containing some instances which were represented in Java classes. We have at least one part of the source code that should trigger the query to detect the code smell and another one that should not.

6.5.2 Detection rules implementation

The next step in the development of code smells detection queries is their implementation in CodeQL. As shown in Figure 6.1, we created a query for each code smell in a *.QL* file, which uses the code smells detection classes, already discussed in Section 6.4. During our static analysis, performed by CodeQL, these queries are called to find the instances of the chosen code smell, based on the extracted CodeQL databases.

To illustrate how our detection queries work, Figure 6.9 presents a query that is called to find occurrences of the "Abusive use of indexes" code smell.

As a reminder, in Section 3.2.2 we explained that queries written in CodeQL are composed of 3 important parts: the *from*, *where* and *select*.

The **from clause** allows us to use the imported CodeQL Libraries with the MongoDB code smells detection classes. In this case, we use the *CreateIndexMethodCall* class that finds every occurrences of an index creation.

The **where clause** is the part where we can write the search conditions. In our example, the collection name, on which the index is created, must not be *"unknown"* – which is the default value returned when we cannot obtain the collection name – and the number of indexes created for this collection must equal or exceed 40, as defined in the *"isTooMuchIndex"* predicate.

```
/**
 * @id AbusiveUseOfIndexes
 * @kind problem
 * @description Detect instances of the Abusive Use of Indexes
   code smell
 */

import codeql_queries.java.CodeSmells.TooManyIndexes.
   TooManyCreateIndex

from CreateIndexMethodCall createIndex, string collectionName,
   string attributeName
where
    collectionName = createIndex.getCollectionName() and
    attributeName = getMostUsedAttribute(collectionName) and
    not collectionName.matches("Unknown") and
    isTooMuchIndex(collectionName)
select
    createIndex, "This index creation could cause performance
   issues because too many indexes ("
    + getTotalCreateIndex(collectionName)
    +  ") have been created for the collection '$@'. This
   collection has "
    + getNumberOfField(collectionName)
    + " fields and the most used field is '$@' ("
    + getTotalUseAttribute(collectionName, attributeName)
    + " times)."
    +"\nTry to get less than 40 indexes by collection !",
    createIndex, collectionName,createIndex.getArgument(0),
   attributeName

```

Figure 6.9: CodeQL query to detect the creation of too many indexes

Finally, the **select clause** is used to display the problematic code artifacts that matches with the conditional clause, and a message to explain the code smell. In the example, we display the code fragment that caused the creation of the index and a message. This message explains why having too many indexes is problematic and describes the context of the smell: the total number of indexes for the collection, the number of fields in the collection, and the field most used to create indexes.

6.5.3 Test of the rules on a real project

The final step is to test our queries on real-world Java projects.

To do this, we first check the project manually to find potential code smells that could be hidden in the code. Then, we run the smell detection queries on the project. If a code smell artifact, that we discovered manually, does not triggers the detection query, then we add this case to the test project and modify the detection query to consider this case during the detection.

Examples

This methodology allowed us to discover, for example, that the official MongoDB driver allows the user to directly execute a query on the database by sending the query as a string literal. This way of interacting was not presented in the documentation we

read, but by looking manually for the “$lookup" string in the program in order to find instances of the code smell *Separating data accessed together*, we discovered that this method was used in some of the projects in our dataset.

A second example of usage that we discovered like this is the fact that older versions of MongoJack did not obtain the MongoDB collections the same way as the newest version for which we read the documentation.

Chapter 7

Application of the analysis on open-source projects and results

In this chapter, we present the application of our tool on real-world Java projects. First, we present the selection process of the projects, introduce some statistics on those projects and their usage of MongoDB databases. Then we present the application of our analysis method to this large set of programs and the results we obtained from this analysis. Finally, we draw our conclusion about these results and the prevalence of MongoDB code smells.

7.1 Context: open-source projects

7.1.1 Projects dataset

To validate our queries and the existence of code smells, we tested our analysis methods on open-source projects. Our thesis being part of a larger research project led by the University of Namur (UNamur) and the Università della Svizzera italiana (USI), we had access to a large dataset that identifies projects and their GitHub repositories. Furthermore, the dataset contained other information such as their programming language, the kind of database they used and their rating on GitHub.
This dataset has been created by [6] for their research about multi-database models in open-source projects. It is based on data provided by *Libraries.io*[1], an open-source web service that helps software developers to keep track of the dependencies they are using inside of a project Therefore it provides an extensive list of projects with their dependencies. The dataset we used contains around 42k projects that rely on one or several database technologies. To ensure a representative sample, Benats et al. excluded smaller or “low-quality" projects and only kept the projects that have a minimum size of 100kB, at least two contributors and that have been starred at least twice. The exact filtering process is explained in greater detail in their paper [6].

7.1.2 Selection of the projects

The selection criteria we used to choose our set of programs are quite simple because we wanted to include as many projects as possible in our test sample. The project had to use Java as a main programming language and use a MongoDB database as one of its database management systems. As explained in Section 7.1.1, the quality of the projects has already been addressed by the filtering criteria of the dataset we used. After this

[1]Libraries.io – `https://libraries.io`

second filtering, where the objective was to only keep Java projects using MongoDB, we ended with a sample of 693 project repositories.

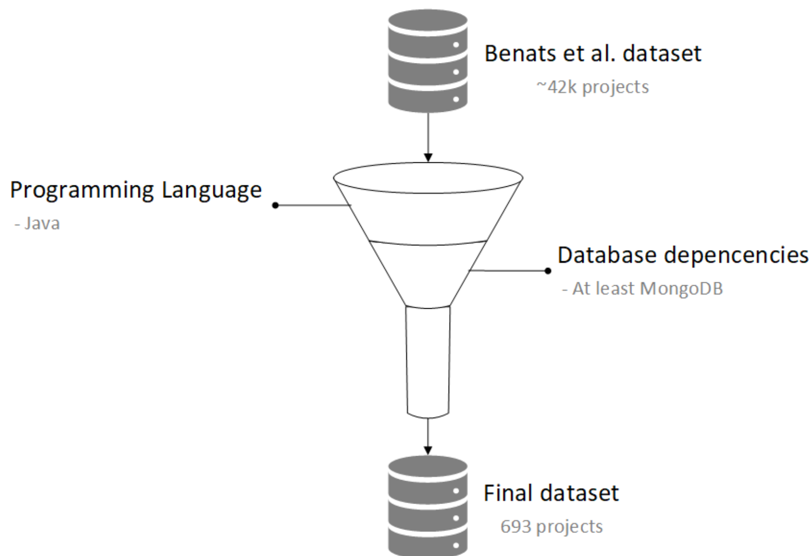

Figure 7.1: First filtering of the projects

7.2 Results

This section presents the results we obtained by applying our analysis method to the projects. First, we analyze the results of the database generation phase in terms of success rate for the compilation of the projects and the generation of the CodeQL database. Then, we analyze results concerning the use of different MongoDB drivers in the selected projects. Finally, we analyze the final results of our code smell detection.

7.2.1 Projects compilation and database creation

Despite the use of the script that automatically extracts the CodeQL databases for the projects, not all the projects were successfully compiled and they did not result in a complete and usable CodeQL database. This was mainly due to the open-source nature of the dataset we analyzed and the fact that each project was cloned from the last commit of the main branch of the repository. Indeed, for some projects, the version pushed to the master branch of a repository contains errors. For others, the resolution of the dependencies leads to errors during the compilation.

Figure 7.2 compares the number of projects that succeeded to the total number of projects. In this figure, we present numbers in terms of subprojects as well as in terms of projects. The previously mentioned projects often contain several programs that need to be generated separately and that are defined by their own project file (pom.xml for Maven projects, build.gradle for Gradle projects, etc.). This means that some projects can contain subprojects that build correctly, and others that do not. So, the analysis has been performed on those subprojects and not on the projects as a whole. To provide a better understanding, we report the subprojects results to projects and provide both numbers.

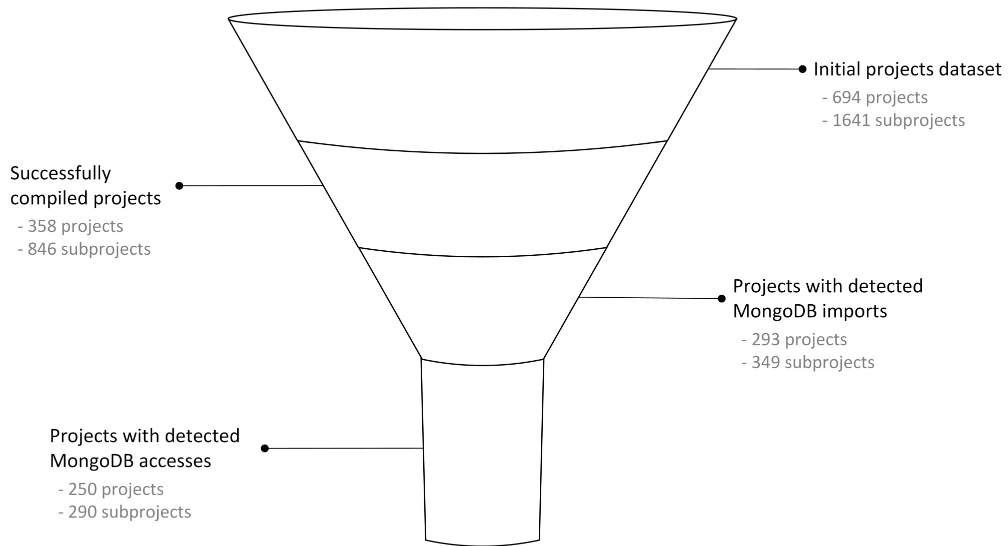

Figure 7.2: Second filtering of the projects

7.2.2 Analysis of the usage of MongoDB drivers

Before presenting our code smell detection results, we provide some numbers about the different drivers that are imported and used by the projects we analyzed. This provides a better understanding of the sample we used. Considering the size and the nature of our projects sample, these numbers could also provide an interesting adoption rate estimation of all the different drivers available.

Import of MongoDB drivers and ODMs

The first analysis we made about the use of MongoDB drivers and ODMs was to check how many programs imported them from our projects, and which drivers/ODMs were imported together. Those results have been reported on the Venn diagrams of Figures 7.3 and 7.4. Those figures respectively present the number of subprojects and projects that import each driver we analyzed. To present all of the most used MongoDB drivers, we also included Morphia, even though we did not implement access and code smell detection for this driver.

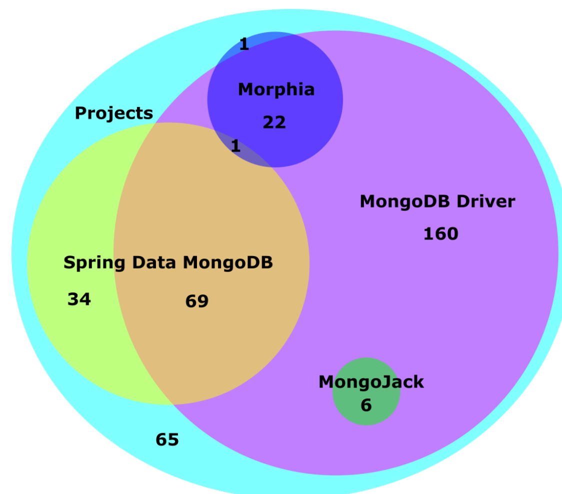

Figure 7.3: Import of MongoDB drivers in projects

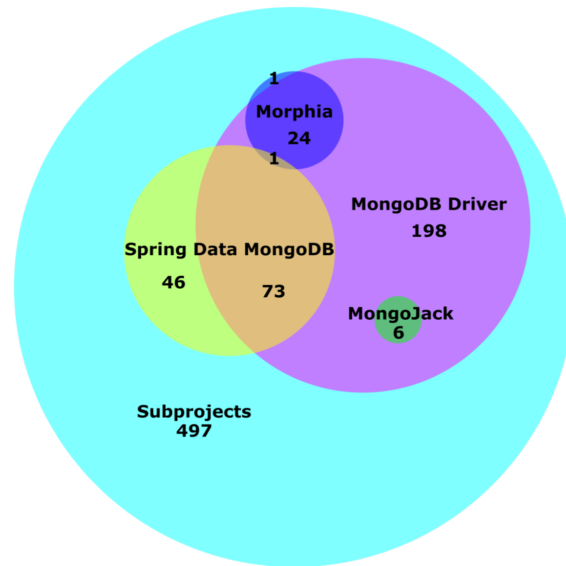

Figure 7.4: Import of MongoDB drivers in subprojects

As we can see in Figure 7.4, a large proportion of subprojects (497/846 subprojects, or 58.75%) are not importing any of the drivers we analyzed. This is probably due to the fact that we considered that every subproject of a project that declared MongoDB in its dependencies is likely to import a MongoDB driver and that all these subprojects should be analyzed. Indeed, when we look at the same number for projects, only 65 projects (18.16%) do not import any of our drivers. This can be explained in two ways. Either they were using MongoDB in a previous version and then changed to another database system, or they are using a driver that we did not include in our research.

When we look at the number of projects that are importing each driver, we can see that the official MongoDB driver is by far the most used, being imported in 258 projects. The second most used driver is Spring Data MongoDB with 104 imports, followed by Morphia with 24 imports and MongoJack with only 6 imports.

It is also interesting to notice that the official MongoDB driver is imported in almost every project that imports Morphia (23/24) and in every projects that imports MongoJack. This might suggest that Morphia and MongoJack both require some access to be done with the MongoDB driver first, or maybe that they reuse some of the classes provides by the official MongoDB driver API. The official MongoDB driver is also imported in projects using Spring Data but in a smaller proportion. We can see that it is imported in two-third of these projects.

Usage of MongoDB drivers and ODMs

A second set of numbers we collected during our study is the number of projects and subprojects in which we detected calls to driver or ODM methods that access the database. As we can see in Figure 7.5 and Figure 7.6, Morphia has not been included in this analysis as we did not implement the accesses extraction for this driver. When comparing Figure 7.5 with 7.3 and Figure 7.6 with Figure 7.4, we can see that the numbers are similar. All the projects that are calling MongoJack methods are also calling methods from the official MongoDB Driver. This confirms that MongoJack requires the use of MongoDB Driver methods to access the database. However, we notice that there are much fewer projects that use Spring Data MongoDB and the official MongoDB driver methods together than projects importing them together. This suggests that Spring Data MongoDB can reuse some classes defined inside the MongoDB driver API but also

that it does not require the developer to use methods from this driver.

We can also notice in Figures 7.5 and 7.6 that we have a larger proportion of projects in which we did not detect any access through our list of MongoDB drivers (30.17%) than projects that did not import any of them (18.16%). This is probably due to the fact that we did not extract accesses made with Morphia. It is also possible that some projects contain unnecessary imports due to a change of database system. Finally, it may also mean that the recall of our access extraction phase could be improved by detecting more drivers or more ways of using the drivers for which we already implemented the access extraction.

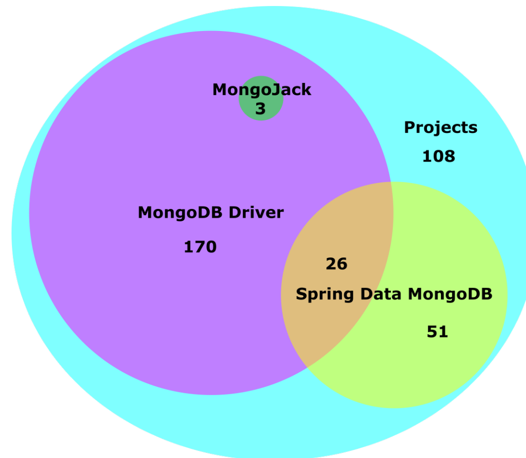

Figure 7.5: Usage of MongoDB drivers in projects

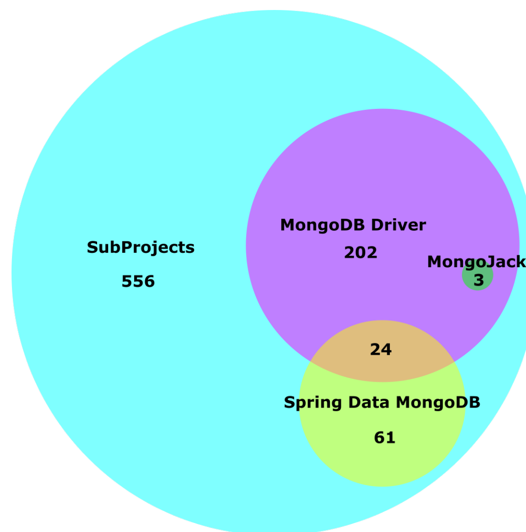

Figure 7.6: Usage of MongoDB drivers in subProjects

7.2.3 Code smells detection

In this section, we present the results we obtained during the code smell detection phase. First, we present our results from a more global point of view, then we give for each code smell an instance example we detected in the open-source projects.

As we can see in Figure 7.7, we detected at least one instance of each code smell in our projects dataset, except for the *Abusive use of indexes*. We can also notice that,

except for the *Too long field names* code smell, for which we detected 162 instances, the number of instances detected remains quite low, with respectively one and four instances. This indicates that the principles of MongoDB that are concerned by the code smells we analyzed were globally well respected for those projects. This low number of detected smells is probably due to the filtering criteria used by Benats et al. [6] while creating their dataset. They indeed focus on good quality projects (see Section 7.1.1). Using less restrictive quality criteria could probably increase the number of smell occurrences and lead to the discovery of new types of code smells.

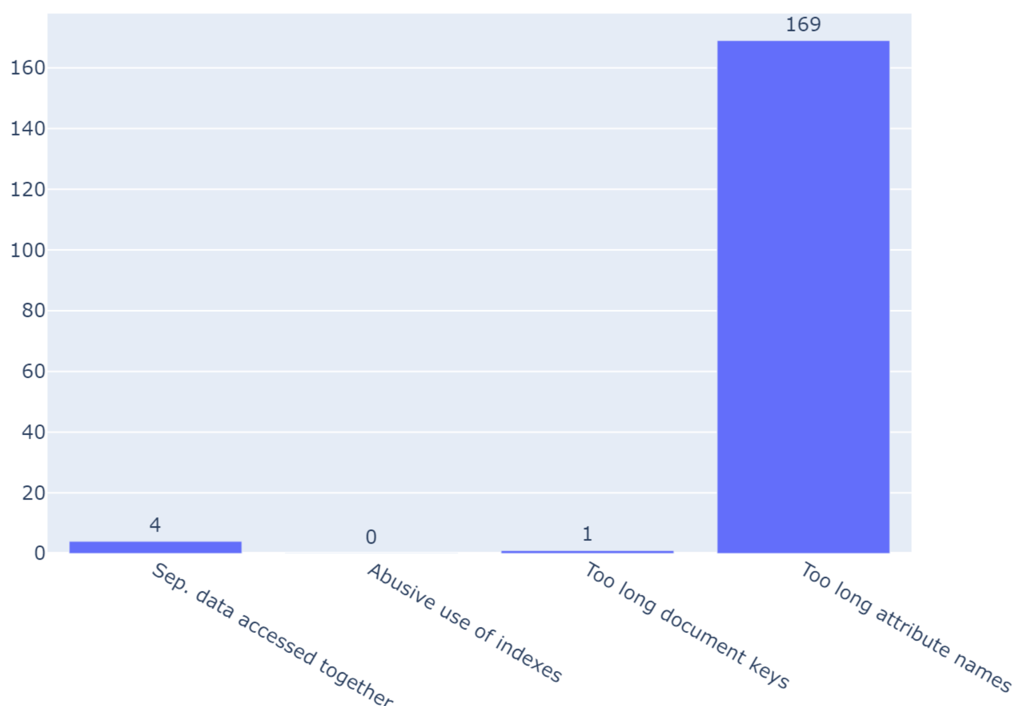

Figure 7.7: Instances detected for each code smell

The graph presented in Figure 7.8 represents the number of projects and subprojects in which we discovered each code smell. As we can see, the number of projects containing the code smell *Too long field names* is significantly lower than the number of code smells instances we discovered. Indeed, most of the instances were contained in only two projects.

Separating data accessed together

With only one project containing this code smell, it is one of the least widespread code smell of our analysis, excluding the *Abusive use of indexes* code smell. This is actually a good sign because this code smell breaks one of the fundamental design rules of MongoDB databases. This means that at least in the projects we analyzed, almost all MongoDB databases are designed and used differently from relational databases. All of the four instances we discovered have been found in the project *Mongodbuserstore* from

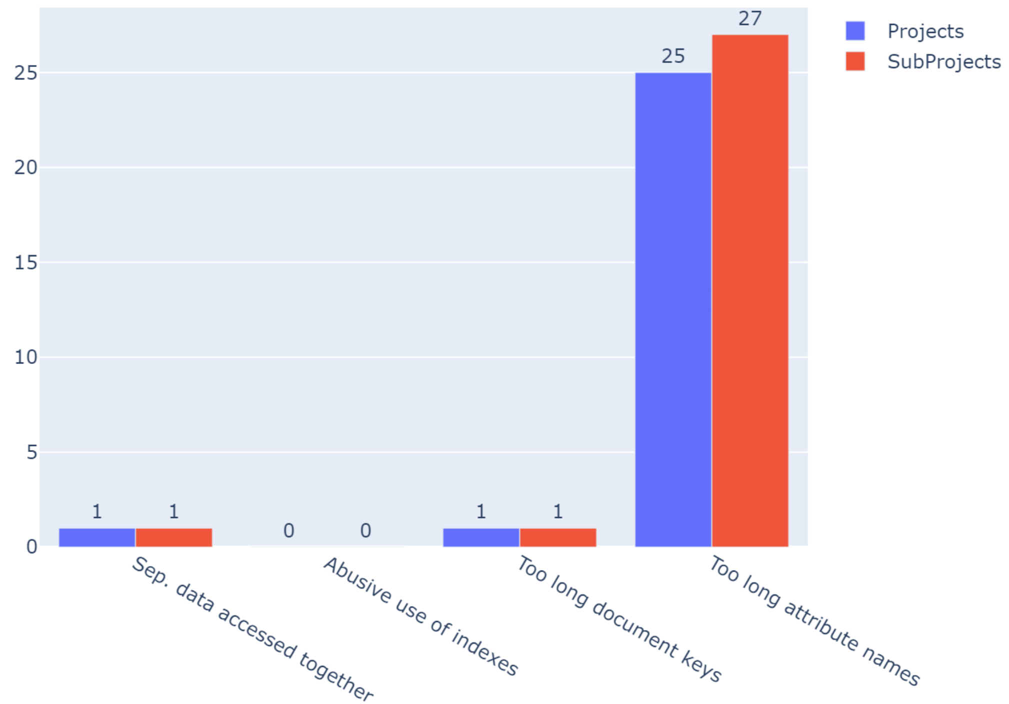

Figure 7.8: Number of projects containing each code smell

asanthamax.[2] This project provides a user-store implementation for MongoDB.

Table 7.1 contains the four instances of collections accessed together by using lookup aggregations. For each instance, it contains the two collections and the number of queries joining them. From the name of the collections and the fact that they are accessed together, we can immediately infer that the database has been designed like a relational database, with collections for users, roles and user attributes, and a relational collection to represent many-to-many relations between roles and users.

Nb Lookups	Collection A	Collection B
7	UM_USER	UM_USER_ATTRIBUTE
7	UM_USER	UM_SHARED_USER_ROLE
2	UM_USER	UM_USER_ROLE
3	UM_ROLE	UM_USER_ROLE

Table 7.1: *Separating data accessed together* instances detected in asanthamax/mongodbuserstore

Figure 7.9 presents one of the instances detected. This instance, as well as the others, consist of a string query where the parameters have been replaced by question marks. Those queries are stored in constants to be reused in the program. We can see here that the query uses a lookup aggregation between the collection UM_ROLE, specified with the *collection* attribute, and the collection UM_USER_ROLE, specified in the *from*

[2] https://github.com/asanthamax/mongodbuserstore

attribute of the *lookup* object.

```
43    private static void setAdvancedProperty(String name, String
      value) {
44      Property property = new Property(name, value, "", null);
45      CUSTOM_UM_ADVANCED_PROPERTIES_TEMP.add(property);
46    }
...   ...
94    setAdvancedProperty("UserRoleMONGO_QUERY", "{'collection' : '
      UM_ROLE',$match : {'UM_TENANT_ID' : '?','userRole.UM_TENANT_ID'
       : '?','users.UM_TENANT_ID' : '?','users.UM_ID' : '?'},'
      $project' : {'UM_ROLE_NAME' : 1,'_id' : 0},'$lookup' : {'from'
      : 'UM_USER_ROLE','localField' : 'UM_ID','foreignField' : '
      UM_ROLE_ID','as' : 'userRole'},'$unwind' : {'path' : '$userRole
      ','preserveNullAndEmptyArrays' : false},'$lookup_sub' : {'from'
       : 'UM_USER','localField' : 'userRole.UM_USER_ID','foreignField
      ' : 'UM_ID','as' : 'users','dependency' : 'userRole'},'
      $unwind_sub' : {'path' : '$users','preserveNullAndEmptyArrays'
      : false}}");
95
```

MongoDBUserStoreConstants.java

Figure 7.9: Example of a detected instance of the *separating data accessed together* smell in asanthamax/mongodbuserstore


```
1     {
2       "ruleId" : "SeparatedDataAccessedTogether",
3       "ruleIndex" : 0,
4       "message" : {
5         "text" : "You have 3 queries that perform a lookup
            operation between the collections UM_ROLE and
            UM_USER_ROLE. Maybe you should consider storing data in
             the same collection to avoid performance issues."
6       },
7       "locations" : [ {
8         "physicalLocation" : {
9           "artifactLocation" : {
10            "uri" : "src/main/java/org/wso2/carbon/mongodb/
                userstoremanager/MongoDBUserStoreConstants.java",
11            "uriBaseId" : "%SRCROOT%",
12            "index" : 1
13          },
14          "region" : {
15            "startLine" : 94,
16            "startColumn" : 9,
17            "endColumn" : 629
18          }
19        }
20      }],
...     ...
26    }
```


Figure 7.10: Analysis output for the smell detected in Figure 7.9

The SARIF output of our analysis corresponding to this instance is presented in

Figure 7.10. The whole object represents the result. The *ruleId* attribute gives the name of the code smell. The *message* property contains information for the user about the code smell detected. It gives the name of the collections that are accessed together as well as the total number of queries accessing those collections. The *location* property provides information about the position of the code smell in the source code, with the path to the file, the line and the column at which the code smell is located. One result object is generated for each query in order to be able to highlight in a potential tool the position of each problematic query.

Too long field names

As mentioned before, the *Too long field names* code smell is the code smell we detected the most in our analysis, with 169 instances in 25 projects. Figure 7.11 shows the number of instances found per project. Projects containing two or fewer instances have been grouped in *"Other projects"*. We can see that four projects contain a large majority of all the detected instances.

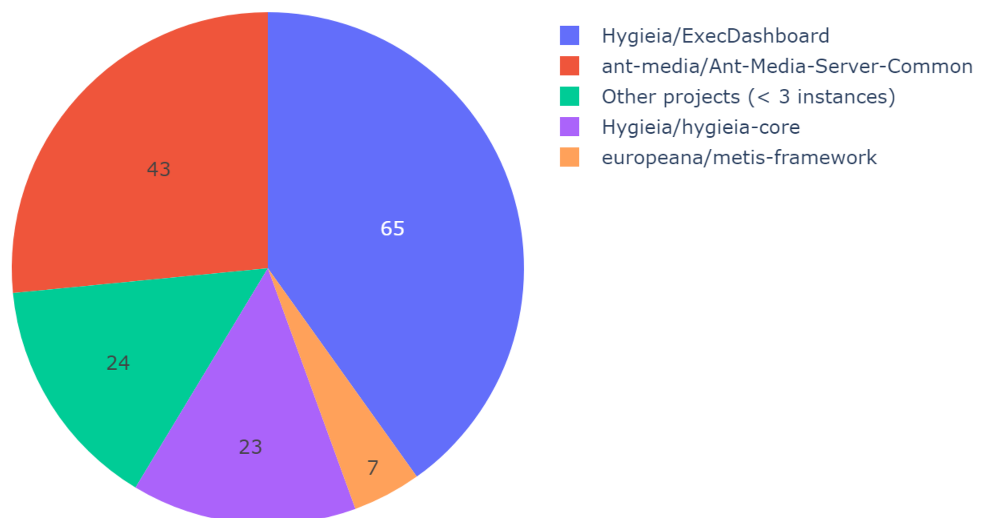

Figure 7.11: *Too long field name* instances per project

A representative example is presented in Figure 7.12. In this case, it has been detected in a class mapped to a collection using Spring Data MongoDB. As we can see, the class is annotated with the *@Document* annotation and mapped with the collection *"article_info"*. Inside this class, there is an attribute named *"articleSecondaryArticleCategoryList"* which is not renamed with any annotation. This means the name of this attribute corresponds to the name of the field in the collection. This example is relevant because the name of this field could have been easily reduced to *"secondaryCategories"*. Furthermore, this class contains information about articles in a web shop, which can quickly represent a long list of documents. The SARIF result object corresponding to this instance is presented in Figure 7.13.

```
22    @Document(collection = "article_info")
23    public class ArticleView extends BaseView {
...     ...
243     @NotNull(message = "...")
244     private JSONArray articleSecondaryArticleCategoryList;
...     ...
472   }
473
```

ArticleView.java

Figure 7.12: Example of a detected instance of the *too long field name* smell in ChuangShiTeam/NowUI-Cloud

```
1    {
2      "ruleId" : "LongFieldName",
3      "ruleIndex" : 0,
4      "message" : {
5        "text" : "Avoid using long Field names as they are stored
             in DB for every document and may waste a lot of space (
             articleSecondaryArticleCategoryList)"
6      },
7      "locations" : [ {
8        "physicalLocation" : {
9          "artifactLocation" : {
10           "uri" : "module-cms/module-cms-sys/src/main/java/com/
                 nowui/cloud/cms/article/view/ArticleView.java",
11           "uriBaseId" : "%SRCROOT%",
12           "index" : 0
13         },
14         "region" : {
15           "startLine" : 244,
16           "startColumn" : 23,
17           "endColumn" : 58
18         }
19       }
20     } ],
...    ...
26   }
```

Figure 7.13: Analysis output for the smell detected in from Figure 7.12

Too long document keys

The only instance of the *Too long document key* code smell has been discovered in the EUMSSI/EUMSSI-platform project. As we can see in the code from Figure 7.14, it consists of the creation of a UUID which is then appended to a BSON document with *_id* as the key, making it the identifier of this document. Because this document is inserted in the *"content_items"* collection later in the source code, it is then considered as an instance of the *Too long document keys* code smell. The SARIF output for this instance is presented in Figure 7.15.

```
47      public static MongoClient mongoClient = new MongoClient();
48      public static MongoDatabase db = mongoClient.getDatabase("
        eumssi_db");
49      public static String _collection = "content_items";
...     ...
208     Document mdoc = new Document();
209     UUID id = UUID.randomUUID();
210     mdoc.append("_id", id);
...     ...
247     db.getCollection(this._collection).insertOne(mdoc);
248
```

EUMSSI_DBQ.java

Figure 7.14: *Too long document keys* instance detected in EUMSSI/EUMSSI-platform


```
1     {
2       "ruleId" : "LongTypeIndexes",
3       "ruleIndex" : 0,
4       "message" : {
5         "text" : "Using UUIDs as a document ID or in an indexed
             field could cause performance issues and large indexes.
              You should consider using a more simple id or an
             ObjectID"
6       },
7       "locations" : [ {
8         "physicalLocation" : {
9           "artifactLocation" : {
10            "uri" : "src/de/l3s/eumssi/news/EUMSSI_DBQ.java",
11            "uriBaseId" : "%SRCROOT%",
12            "index" : 0
13          },
14          "region" : {
15            "startLine" : 210,
16            "startColumn" : 8,
17            "endColumn" : 30
18          }
19        }
20      } ],
...     ...
26    }
```


Figure 7.15: Analysis output for the smell detected in Figure 7.14

Abusive use of indexes

For this last code smell, our analysis returned one positive result in the bwaldvogel/mongo-java-server project. However, this case cannot be considered as an instance of the code smell. After a manual verification, we discovered that the project in question is an in-memory MongoDB simulation and that the instance came from a test class. Such cases were anticipated by excluding test packages from the analysis, but in this case, the whole subproject was dedicated to testing, and therefore not the package only. Even if it cannot be considered as an instance of the code smell, we still present this result to provide an example of what our implementation is able to detect and the output it generates.

```
22      public abstract class AbstractBackendTest extends AbstractTest
        {
...       ...
3801      public void testCompoundSparseUniqueIndexOnEmbeddedDocuments
        () throws Exception {
3802        collection.createIndex(json("'a.x': 1, 'b.x': 1"), new
        IndexOptions().unique(true).sparse(true));
...       ...
3832      }
...       ...
6306    }
6307
```

AbstractBackendTest.java

Figure 7.16: Example of an index creation detected in bwaldvogel/mongo-java-server


```
1    {
2      "ruleId" : "TooManyCreateIndex",
3      "ruleIndex" : 0,
4      "message" : {
5        "text" : "This index creation could cause performance
             issues because too many indexes (47) have been created
             for the collection '[testcoll](1)'. This collection has
              135 fields and the most used field is '[c](2)' (6
             times).\nTry to get less than 40 indexes by collection
             !"
6      },
7      "locations" : [ {
8        "physicalLocation" : {
9          "artifactLocation" : {
10           "uri" : "test-common/src/main/java/de/bwaldvogel/mongo
                /backend/AbstractBackendTest.java",
11           "uriBaseId" : "%SRCROOT%",
12           "index" : 0
13         },
14         "region" : {
15           "startLine" : 3802,
16           "startColumn" : 9,
17           "endColumn" : 105
18         }
19       }
20     } ],
...    ...
60   }
```


Figure 7.17: Analysis output for the example from Figure 7.16

Figure 7.16 presents one of the 47 indexes creations on the same collection detected in bwaldvogel/mongo-java-server. The SARIF analysis output for the first of these index creations is presented on Figure 7.17. As we can see in the message object, we warn the programmer that there might be too many indexes on this collection. In addition to this, we also give information about the number of fields in this collection and the most used one. The objective of giving these details is to help the developer making the decision himself if there are really too many indexes. The collection on which the index is created

is automatically initialized before each test with the name testcoll thanks to a method declared in the AbstractTest method that the AbstractBackendTest class extends.

Chapter 8

Limitations, improvements and future works

In this chapter we list limitations we have identified in our thesis, as well as the potential improvements and some future works to be realized in the field of NoSQL code smells. The objective is to provide different tracks to enrich our taxonomy and to improve the tool that we have implemented.

8.1 Improvements of the code smell taxonomy

In this section, we discuss potential improvements for our taxonomy.

8.1.1 Discover more code smells

First of all, our taxonomy does not pretend to be exhaustive. Indeed, we did not consider all the code smells that exist in MongoDB since our work also involved the creation of code smells detection methods and we only had a limited time available.

Moreover, MongoDB is a relatively recent DBMS and, to our knowledge, there is currently no scientific research made on code smells in MongoDB. Nevertheless, as NoSQL becomes a DBMS that starts to be more widespread and takes a more important place in scientific research, maybe some more research on code smells in MongoDB will start to emerge. Therefore, with the help of this new research and other blogs or forums, it will be interesting to extend the search to find other code smells based on the methodology we provided in Section 4.1.

We also thought of another approach to find new code smells in MongoDB. This one would consist in the realization of performance tests on the interactions of Java project with a MongoDB database. Indeed, the very principle of NoSQL databases is to optimize the speed of queries. However, if some queries seem to be less efficient than others, then it could mean that we are dealing with code smells in MongoDB.

Whatever approach is used, we consider that the taxonomy we have provided in Appendix A could be a good starting point to continue the research on code smells in MongoDB.

8.1.2 Impact severity metrics

Currently, our taxonomy does not provide information about the severity of code smells. Indeed, if we deduced logically the kind of impact they would have on a system, their real impact has not been measured yet. Such information could help the programmer to prioritize the code smells he should address as a priority.

8.2 Improvements of the analysis tool

In this section, we explain the different limitations of our code smells analysis tool and the improvements that could be done to resolve them. These improvements include the creation of an executable tool, the use of schema inference to detect other code smells, the improvement of the CodeQL database extractor, and the inclusion of other ODMs or drivers for analysis.

8.2.1 All-in-one tool creation and IDE integration

Currently, each step of the code smells detection is performed manually one after the other. However, the ideal would be to have an executable tool that would perform the different steps presented in Section 6. This tool could take as input the source code to be analyzed and output the code smell instances.

Another solution would be to present this tool as an extension or plugin of development software like Eclipse or Visual Studio. This would allow to directly ensure that the program, which is developed on one of these IDE, does not contain code smells.

8.2.2 Use of schema inference

Some code smells cannot be detected based on a simple observation of the queries made to the database. In some cases, it might be necessary to know how the data is stored in the database. Therefore, the idea would be to infer the MongoDB database schema based on the different queries. This inference can be made in different ways, depending on the ODM or the driver.

Spring data and Morphia: This is the easiest way to infer about the database schema. Indeed, since both ODMs rely on entities to represent MongoDB documents and collections, we only need to observe these entities to infer on the schema.

MongoDB Java driver: It is a bit more complex to infer the schema of the MongoDB database because the collections and the documents are not necessarily represented by classes in Java. In this case, to make the inference, we would have to observe the queries that are sent to this database. Indeed, the fields used by the CRUD queries would be good sources of information about the database content.

MongoJack: For MongoJack, it is possible to rely on the two techniques presented just before. MongoJack allows to map collections or documents using entity classes but also allows to perform queries directly on the collection by transferring BSON document objects. It would thus be necessary to observe the entity classes that represent the collections, if they exist, or to analyze the content of the CRUD queries.

8.2.3 Improvements of the CodeQL database extractor

Currently, the database extractor does not solve all the limitations that were discussed in Section 6.1.3. Therefore, in this section, we explain different ways to improve the database extraction script.

Delombock projects using Lombock dependencies: As explained in the limitations of CodeQL, some projects are using Lombock dependencies, which causes some files not to be generated when extracting from the CodeQL database. Therefore, when

extracting the database, it would be interesting to detect automatically if the project has a Lombock dependency or not. If the project uses Lombock, it would be interesting to replace the Lombock annotations with corresponding lines of code to be parsed by CodeQL. To do this, as explained on the Lombock site[1], this library offers the ability to *Delombock* files using Lombock annotations with a command line. So, before executing the extraction of the CodeQL database, it would be necessary to detect the files using Lombock annotations and, for each of these files, to execute the appropriate command line according to the Framework used by the project.

Try compilation on other project versions: As we explained in Section 6.1.3, some project versions may contain errors and could cause compilation problems. As a reminder, the projects used in our research are open-source projects stored on GitHub repositories. Moreover, we are currently using the latest commit on the master branch of the repository to perform our analysis. When the compilation of a project version fails, it would be interesting to try to compile it with another version of the project contained in the repository such as the latest release version for instance. This would allow finding the first most stable version.

8.2.4 Include project analysis using other ODMs or drivers

Currently, our code smells analysis tool can only analyze projects using MongoDB Java driver, Spring data and MongoJack. The ideal would be to implement the analysis for other ODMs or drivers like Morphia, which we have already discussed previously.

8.3 Future works

In this section, we provide some potential avenues to deepen the research in the field of code smells in NoSQL databases interactions.

8.3.1 Deeper analysis about the usage of NoSQL databases

NoSQL databases, despite the fact that it is becoming quite popular nowadays, have not been studied a lot in the scientific domain. A deeper analysis of the way programmers use them in their systems and the way they interact with them would be interesting to study. A better understanding of their usage could lead to better tools, and in our case in the detection of other anti-patterns or code smells.
For example, a deeper analysis could be carried on by using schema inference – which we have already discussed in Section 8.2.2 – to show how the flexibility of the schema is used in the studied system.

8.3.2 Empirical studies about the impact of code smells

As already said in Section 8.1.2, the impact of the code smells we defined have not been studied yet. It would thus be interesting to conduct empirical studies to measure their real impact on system performance and maintainability. A better understanding of their impact would enhance the taxonomy we proposed by allowing to determine the severity of each smell.

This empirical study could be achieved for example by performing performance tests on real-world systems before and after the resolution of existing code smells, or by introducing voluntarily code smells inside "healthy" systems to measure the loss in performance.

[1] `https://projectlombok.org/features/delombok`

8.3.3 Visualization of the code smells

Another idea would be to provide a proper visualization method to present code smell instances detected inside of a program. A good visualization is key to the understanding of data, and would therefore help in the maintenance of systems that contains code smells. When we think about the visualization of code smells, we could imagine visualizations that indicates the parts of a system that contains the most code smells and that would need refactoring in priority.

8.3.4 Code smells in other types of NoSQL databases

Finally, our thesis focuses only on MongoDB, which belongs to the family of document databases. However, the NoSQL databases appellation covers a lot of other database families such as key-value stores, column-oriented databases and graph databases. Each kind of database has its own peculiarity and therefore its own potential code smells. Investigating code smells for other database systems than MongoDB is then necessary, as we cannot systematically transpose the ones we defined in this thesis to other database systems.

Chapter 9

Conclusion

In the introduction, we first presented the context of our thesis as well as our motivations and objectives. We have highlighted the motivation of studying code smells to improve code and database interactions quality, as well as the advantages of establishing a taxonomy to better understand them. We then briefly introduced the rationale of static analysis for the detection of these code smells. Indeed, it allows us to detect instances of code smells without requiring the execution of the program. We ended this introduction with an overview of our thesis structure.

To begin our research, in Chapter 2, we established a state of the art, presenting the previous researches in the domains used in our thesis. We presented code analysis, with an emphasis on static analysis methods like abstract syntax trees, data-flow and control-flow analysis. We then turned to the researches about code smell, beginning with the origin of the term and a first definition. We spoke about the influence of those code smells on object-oriented programming and in SQL databases interactions. Once those code smells were defined, we focused on the existing methods and tools to detect them, both in Java code and in SQL database interactions. We finished our state of the art by exploring researches about code smells for NoSQL systems, which led us to the fact that no code smells have yet been scientifically defined or analyzed.

In Chapter 3, we introduced the background of our research, by presenting MongoDB, i.e. the database system we used, and CodeQL, i.e the static analysis tool that we used to detect code smells.

In Chapter 4, we started filling the research gap we found by defining and following a methodology to create a first list of 11 code smells for MongoDB databases interactions. We provided code examples for each of these code smells and we hypothesized their potential impact on the system. Once this list had been created, we classified those smells into three groups: *Relational design ghosts*, *Human-oriented decisions* and *Design oversights*. These three groups form the base of our taxonomy. Afterwards, we took inspiration from existing code smell taxonomies to improve and complete our own.

Chapter 5 was dedicated to the design of a tool that allowed us to detect MongoDB code smells in Java projects. We justified the choice of static code analysis for the creation of this tool and presented the different phases of our analysis process: the MongoDB access extraction – which allowed us to locate and collect information about the accesses to a MongoDB database inside Java code, and the code smell detection phase – which uses detection rules and the previously extracted accesses to locate instances of code smells. Then, we selected the code smells for which we designed detection rules based on selection criteria.

The implementation of the previously defined process took place in Chapter 6. We first explained the choice of CodeQL as a static analysis tool among other alternatives like SpotBugs or Spoon. We then described the way we used it to perform analysis steps

like database accesses extraction and code smell detection.

Once implemented, our tool was tested and applied on a large number of projects in Chapter 7. We began this chapter by the presentation and the selection of the Java projects on which we performed our analysis. We then explained the way we applied our tool to perform the analysis on such a large dataset. Finally, we exposed the results we obtained for each code smell, providing some numbers as well as examples of instances we detected and the corresponding analysis result. Thanks to numbers gathered in the MongoDB access extraction phase, we were able to provide an overview of the different MongoDB drivers use.

Finally, Chapter 8 gathers the limitations we faced during our work. For each of these limitations, we have tried to give some hints for future improvements that could be realized. These improvements concern the taxonomy that we have developed and the code smells analysis tool that we have implemented. We also conclude this chapter with some future works that could be realized in the field of NoSQL code smells.

To conclude, we can say that this thesis is a first step in the research field of NoSQL code smells, and more precisely of MongoDB databases interaction code smells. It has successfully proven that such code smells exist, that they are present in open-source projects and that they can be detected with static analysis. There is of course room for improvement, both in the taxonomy we proposed, and in our analysis method, for example in the precision and recall it provides, which we did not evaluate. However, we hope that the basis we have provided will encourage the research in this field.

Bibliography

[1] Elvira Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. A method for assessing class change proneness. In *ACM International Conference Proceeding Series*, volume Part F1286, pages 186–195. Association for Computing Machinery, June 2017. ISBN 9781450348041. doi: 10.1145/3084226.3084239.

[2] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 56, pages 21–225, 2016. ISBN 9783959770149. doi: 10.4230/LIPIcs.ECOOP.2016.2.

[3] Nathaniel Ayewah, David Hovemeyer, David J. Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. ISSN 07407459. doi: 10.1109/MS.2008.130.

[4] Thomas Ball. The concept of dynamic analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1687 LNCS, pages 216–234. Springer Verlag, 1999. ISBN 3540665382. doi: 10.1007/3-540-48166-4_14.

[5] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Does measuring code change improve fault prediction? In *ACM International Conference Proceeding Series*, New York, New York, USA, 2011. ACM Press. ISBN 9781450307093. doi: 10.1145/2020390.2020392.

[6] Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. An empirical study of (multi-) database models in open-source projects. In *Proceedings of the 40th International Conference on Conceptual Modeling (ER 2021)*, 2021.

[7] Tse Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings - International Conference on Software Engineering*, pages 1001–1012. IEEE Computer Society, May 2014. doi: 10.1145/2568225.2568259.

[8] Tse Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder 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):1148–1161, December 2016. ISSN 00985589. doi: 10.1109/TSE.2016.2553039.

[9] Rick Copeland. *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database.* O'Reilly Media, Inc., 2013. ISBN 978-1-449-34004-9.

[10] Daniel Coupal and Ken W. Alger. Building with patterns: The subset pattern | mongodb, March 2019. URL `https://www.mongodb.com/blog/post/building-with-patterns-the-subset-pattern`.

[11] Daniel Coupal and Ken W. Alger. Building with patterns: The extended reference pattern | mongodb, March 2019. URL `https://www.mongodb.com/blog/post/building-with-patterns-the-extended-reference-pattern`.

[12] Phil Factor. *SQL Code Smells*. Simple Talk Publishing, 2014.

[13] Phil Factor. 14 things i wish i'd known when starting with mongodb, September 2018. URL `https://www.infoq.com/articles/Starting-With-MongoDB/`.

[14] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *IEEE International Conference on Software Maintenance, ICSM*, pages 519–520, 2007. ISBN 1424412560. doi: 10.1109/ICSM.2007.4362679.

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings. In *Proceedings - International Conference on Software Engineering*, pages 1037–1039, 2011. ISBN 9781450304450. doi: 10.1145/1985793.1985989.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[17] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Data schema does matter, even in nosql systems! *Proceedings - International Conference on Research Challenges in Information Science*, 2016-Augus, 2016. ISSN 21511357. doi: 10.1109/RCIS.2016.7549340.

[18] Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva, and Upaang Saxena. Nosql databases: Critical analysis and comparison. In *2017 International Conference on Computing and Communication Technologies for Smart Nation, IC3TSN 2017*, volume 2017-Octob, pages 293–299. Institute of Electrical and Electronics Engineers Inc., February 2018. ISBN 9781538606278. doi: 10.1109/IC3TSN.2017.8284494.

[19] Almas Hamid, Muhammad Ilyas, Muhammad Hummayun, and Asad Nawaz. A comparative study on code smell detection tools. *International Journal of Advanced Science and Technology*, 60:25–32, November 2013. ISSN 20054238. doi: 10.14257/ijast.2013.60.03.

[20] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016*, pages 59–69. Association for Computing Machinery, Inc, May 2016. ISBN 9781450341783. doi: 10.1145/2897073.2897100.

[21] Abdullahi Abubakar Imam, Shuib Basri, Rohiza Ahmad, Junzo Watada, Maria T. Gonzlez-Aparicio, and Malek Ahmad Almomani. Data modeling guidelines for nosql document-store databases. *International Journal of Advanced Computer Science and Applications*, 9(10):544–555, 2018. ISSN 21565570. doi: 10.14569/IJACSA.2018.091066.

[22] Github Inc. Codeql documentation, 2021. URL https://codeql.github.com/docs/.

[23] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2006, pages 258–263, 2006. ISBN 0769525741. doi: 10.1109/SP.2006.29.

[24] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming.* Pragmatic Bookshelf, 1st edition, 2010. ISBN 1934356557.

[25] Marc Kenig. Big data anti-patterns, August 2019. URL https://www.linkedin.com/pulse/big-data-anti-patterns-marc-kenig/.

[26] Foutse Khomh, Massimiliano Di Penta, and Yann Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 75–84, 2009. ISBN 9780769538679. doi: 10.1109/WCRE.2009.28.

[27] Foutse Khomh, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, June 2012. ISSN 13823256. doi: 10.1007/s10664-011-9171-y.

[28] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems.* O'Reilly Media, Inc., 2017. ISBN 9781491903063.

[29] Christian Kvalheim. *The Little Mongo DB Schema Design Book.* CreateSpace Independent Publishing Platform, 2015. ISBN 978-1517394028.

[30] Elvis Ligu, Alexander Chatzigeorgiou, Theodore Chaikalis, and Nikolaos Ygeionomakis. Identification of refused bequest code smells. In *IEEE International Conference on Software Maintenance, ICSM*, pages 392–395, 2013. doi: 10.1109/ICSM.2013.55.

[31] Santo Lombardo, Elisabetta Di Nitto, and Danilo Ardagna. Issues in handling complex data structures with nosql databases. *Proceedings - 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012*, pages 443–448, 2012. doi: 10.1109/SYNASC.2012.59.

[32] Panagiotis Louridas. Static code analysis. Technical Report 4, Romanian-American University, Bucarest, December 2006.

[33] Yingjun Lyu, Ali Alotaibi, and William G.J. Halfond. Quantifying the performance impact of sql antipatterns on mobile applications. In *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pages 53–64. Institute of Electrical and Electronics Engineers Inc., September 2019. ISBN 9781728130941. doi: 10.1109/ICSME.2019.00015.

[34] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. *IEEE International Conference on Software Maintenance, ICSM*, pages 381–384, 2003. doi: 10.1109/icsm.2003.1235447.

[35] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. In *Empirical Software Engineering*, volume 11, pages 395–431, September 2006. doi: 10.1007/s10664-006-9002-8.

[36] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iplasma:an integrated platform for quality assessment of object-oriented design. *Proceedings of the 21st IEEE International Conference on Software Maintenance ICSM 2005*, pages 77–80, 2005.

[37] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR)*, page 6. Citeseer, 2006.

[38] Vincent Mathieu. Outils d'analyse statique. Technical report, Université Laval, Québec, 2001.

[39] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: Clone refactoring. In *Proceedings - International Conference on Software Engineering*, pages 613–616. IEEE Computer Society, May 2016. ISBN 9781450341615. doi: 10.1145/2889160.2889168.

[40] Jason McCay. Mongodb indexing best practices - compose articles, 2003. URL `https://www.compose.com/articles/mongodb-indexing-best-practices/`.

[41] Loup Meurice and Anthony Cleve. Supporting schema evolution in schema-less nosql data stores. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 457–461, March 2017. doi: 10.1109/SANER.2017.7884653.

[42] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9694, pages 491–506. Springer Verlag, 2016. ISBN 9783319396958. doi: 10.1007/978-3-319-39696-530.

[43] MongoDB. Mongodb acid transactions whitepaper, n.d. URL `https://www.mongodb.com/collateral/mongodb-multi-document-acid-transactions`.

[44] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, 2007. doi: 10.1109/SCAM.2007.31.

[45] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pages 327–338, 2020. doi: 10.1145/3379597.3387467.

[46] Csaba Nagy and Anthony Cleve. Sqlinspect: A static analyzer to inspect database usage in java applications. In *Proceedings - International Conference on Software Engineering*, pages 93–96, New York, NY, USA, May 2018. ACM. ISBN 9781450356633. doi: 10.1145/3183440.3183496.

[47] Alexander S. Novikov, Alexey N. Ivutin, Anna G. Troshina, and Sergey N. Vasiliev. The approach to finding errors in program code based on static analysis methodology. In *2017 6th Mediterranean Conference on Embedded Computing, MECO 2017 - Including ECYPS 2017, Proceedings*. Institute of Electrical and Electronics Engineers Inc., July 2017. ISBN 9781509067411. doi: 10.1109/MECO.2017.7977127.

[48] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pages 390–400, 2009. ISBN 9781424448418. doi: 10.1109/ESEM.2009.5314231.

[49] Thanis Paiva, Amanda Damasceno, Juliana Padilha, Eduardo Figueiredo, and Claudio Sant'Anna. Experimental evaluation of code smell detection tools. *3th Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015)*, 2015.

[50] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, June 2018. ISSN 15737616. doi: 10.1007/s10664-017-9535-z.

[51] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[52] PMD. Pmd, n.d. URL `https://pmd.github.io/`.

[53] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques, November 2015. ISSN 20477481.

[54] Ghulam Rasool and Zeeshan Arshad. A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering*, 42(2):483–506, 2017. ISSN 21914281. doi: 10.1007/s13369-016-2238-8.

[55] Riak. Nosql databases explained | what is a nosql database | riak, 2021. URL `https://riak.com/resources/nosql-databases/index.html?p=9937.html`.

[56] Lauren Schaefer. What is nosql? nosql databases explained, 2020. URL `https://www.mongodb.com/nosql-explained`.

[57] Lauren Schaefer. 3 things to know when you switch from sql to mongodb, October 2020. URL `https://developer.mongodb.com/article/3-things-to-know-switch-from-sql-mongodb`.

[58] Lauren Schaefer and Daniel Coupal. Separating data that is accessed together, 2020. URL `https://developer.mongodb.com/article/schema-design-anti-pattern-separating-data`.

[59] Lauren Schaefer and Daniel Coupal. Unnecessary indexes, May 2020. URL `https://developer.mongodb.com/article/schema-design-anti-pattern-unnecessary-indexes`.

[60] Spotbugs. Spotbugs, n.d. URL `https://spotbugs.github.io/`.

[61] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *IEEE International Conference on Software Maintenance, ICSM*, pages 270–279, 2013. doi: 10.1109/ICSM.2013.38.

[62] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, October 2011. ISSN 01641212. doi: 10.1016/j.jss.2011.05.016.

[63] TutorialFor. Nosql anti-pattern-document database, 2020. URL `https://www.tutorialfor.com/blog-198278.htm`.

[64] Huib Van Den Brink, Rob Van Der Leek, and Joost Visser. Quality assessment for embedded sql. In *SCAM 2007 - Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 163–170, 2007. ISBN 0769528805. doi: 10.1109/SCAM.2007.23.

[65] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, October 2013. ISSN 01641212. doi: 10.1016/j.jss.2013.05.007.

[66] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *IEEE International Conference on Software Maintenance, ICSM*, pages 306–315, 2012. ISBN 9781467323123. doi: 10.1109/ICSM.2012.6405287.

[67] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings - International Conference on Software Engineering*, pages 682–691, 2013. ISBN 9781467330763. doi: 10.1109/ICSE.2013.6606614.

[68] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings - International Conference on Software Engineering*, pages 800–810. IEEE Computer Society, May 2018. ISBN 9781450356381. doi: 10.1145/3180155.3180194.

[69] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006. ISSN 00985589. doi: 10.1109/TSE.2006.38.

[70] Zeineb Zhioua, Stuart Short, and Yves Roudier. Static code analysis for software security verification: Problems and approaches. In *Proceedings - IEEE 38th Annual International Computers, Software and Applications Conference Workshops, COMPSACW 2014*, pages 102–109. Institute of Electrical and Electronics Engineers Inc., September 2014. ISBN 9781479935789. doi: 10.1109/COMPSACW.2014.22.

Appendices

Appendix A

MongoDB bad smells taxonomy

Smell type	Smell name	Description	Impact type	In-code detection	Gran.	Intra.	VBMP	IO	NR
Relational design ghosts	Separating data accessed together	When data is separated into different collections (for 1-1 or 1-N relation) instead of being embedded in a single document	Performance	Lookup operation between two collections	Method	No	Yes	No	No
	Use of relational collections	When a relational collection is used to represent N-N relationships	Performance	Lookup operation between three collections. Or an entity class that contains only id references.	Class /Method	Yes	Yes	No	No
	Storage of empty values	When Null or Undefined values are stored in a collection instead of not storing the attribute	Performance /Storage waste	Insertion of new documents with a null value or an empty string for a field or update of a document field with a null value or an empty string	Method	Yes	Yes	No	No
	Relying on transaction	When transactions are frequently used, this may indicate that data accessed together is separated.	Performance	When same collections are often accessed between the start and the commit of a transaction	Method	No	Yes	No	No

Human oriented decisions	Too long attribute names	When attributes are stored with long name	Storage waste	With ODM, by checking the annotation on a field or the field name of entity classes. Otherwise, by checking the field used in the database access query of the program	Class /Method	Yes	No	No	No
	Human readable values	When values are stored in a human-readable format rather than in an optimized format	Storage waste	Checking value insertions or updates that correspond to typical representations of data in human format	Method	Yes	No	Yes	No
	Too long document keys	When using long ids value (like UUID type or long string value)	Performance /Storage waste	For UUIDs, by checking the type declared for the identifier in entity classes. Otherwise, by checking the value inserted for the id field of a new document	Class /Method	Yes	No	No	No
Design oversights	Inconsistent order of attributes inside a collection	When consistency between documents is not maintained	Incomplete results	In new document insertions, field ordering is diffrent for a same collection	Method	No	No	No	No
	Databases access spread across the system	When database accesses are spread across the system instead of being group in folders or files	Maintainability	By comparing the number of methods that have one or more database accesses, to the total number of methods in a system or by comparing all accesses locations to ensure that they are grouped in the same place within the file system	System	Yes	No	No	Yes
	Storage of easily calculated values	When easy-calculated values are stored in a collection attribute instead of being calculated in the query or in the program	Performance /Storage waste	Values inserted in a document, which are calculated on the basis of other fields from this document.	Method	No	No	Yes	No
	Abusive use of indexes	When many indexes are created for a single collection and when it is not necessary	Performance	Total index creation in source code for each collection	Class /Method	No	Yes	No	No

Table A.1: MongoDB code smells taxonomy