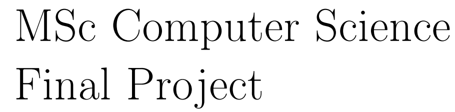

MSc Computer Science
Final Project

Data access paradigms in enterprise software development

Jelle Hulter

Supervisor: Maarten Mulders, Fernando Castor

October, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction **1**
- 1.1 Problem statement & motivation . . . 1
- 1.2 Research Goals and Questions . . . 2
- 1.3 Outline . . . 3

2 Background **4**
- 2.1 Quality attributes . . . 4
- 2.2 Object-Relational Mapping . . . 4
- 2.3 Implementation approaches of frameworks . . . 5
- 2.4 Performance . . . 6
  - 2.4.1 Efficiency of query . . . 7
  - 2.4.2 Overall performance . . . 7
  - 2.4.3 RAPL . . . 8
- 2.5 Code readability . . . 8
  - 2.5.1 Halstead complexity metrics . . . 10
  - 2.5.2 Embedded complexity metrics . . . 11

3 Frameworks **13**
- 3.1 Framework selection . . . 13
- 3.2 Evaluation . . . 13
- 3.3 Ebean . . . 16
- 3.4 Exposed . . . 17
- 3.5 Hibernate . . . 19
- 3.6 JDBI . . . 21
- 3.7 jOOQ . . . 22
- 3.8 MyBatis . . . 23
- 3.9 QueryDSL . . . 25
- 3.10 Spring Data . . . 27
- 3.11 SQLDelight . . . 28

4 Paradigms **30**
- 4.1 Declarativeness . . . 30
- 4.2 Abstraction . . . 30
- 4.3 Relation . . . 31
- 4.4 Implementations of the frameworks . . . 31
- 4.5 Concepts . . . 32
  - 4.5.1 Specification of data access operations . . . 32
  - 4.5.2 Database agnosticism . . . 33
  - 4.5.3 Domain metamodel . . . 34

4.6 Paradigm overview 34

5 Experiments 36
5.1 Test business case 36
5.1.1 Data access operations 37
5.2 Experiment I: Performance 39
5.2.1 Experiment design 39
5.2.2 Experimental Materials 39
5.2.3 Hypotheses, parameters and variables 39
5.2.4 Methodology 39
5.2.5 Execution 40
5.2.6 Analysis 40
5.3 Experiment II: Code Readability 47
5.3.1 Experiment design 47
5.3.2 Hypotheses, parameters and variables 47
5.3.3 Methodology 48
5.3.4 Execution 49
5.3.5 Analysis 49

6 Discussion 54
6.1 Experiment I: Performance 54
6.2 Experiment II: Code Readability 55
6.3 Threats to validity 55
6.3.1 Internal validity 55
6.3.2 External validity 55
6.3.3 Construct validity 56

7 Conclusion 57
7.1 Research Question 1 57
7.2 Research Question 2 58
7.3 Research Question 3 58
7.4 Future work 59
7.4.1 Additional frameworks 59
7.4.2 Non-relational database management systems 59
7.4.3 Additional business cases and operations 59
7.4.4 Discover additional concepts 60
7.4.5 Embedded language metrics 60
7.4.6 Human evaluation 60

Abstract

Persistence of data is an important aspect when developing an enterprise software applications. For storing this data in an persistent manner, database management systems are used. In order for a software engineer to interact with a relational database management system, a language like SQL is used commonly in order to interact with the database. In order to reduce boilerplate and repetitive code, many different data access frameworks have been introduced over time. Each of these frameworks have different properties and have different ways of expressing the data access operation that needs to be executed. Choosing which framework to use for a new enterprise software project is often a critical design choice when implementing a software architecture. In order to help a software architect choose the right data access framework to use, in this thesis, we try to introduce and identify these frameworks into various paradigms. We discovered that many of the data access frameworks have different approaches of how to apply them in practice. Consequently, we have conducted two experiments on the different implementations to see whether we can identify a significant difference between the identified paradigms and their implementations. The first experiment tested the performance of the different implementations and the second experiment focused on the readability of the code of an implementation. These experiments were conducted by implementing the same test business case for each of the 17 implementation approaches. Then, we analyzed the execution time and energy consumption of these implementations, and calculated code complexity in order to reason about the readability of the code. The results have shown that there is no significant difference between the different paradigms in either performance or readability. However, this thesis does give a software architect an overview of the available frameworks, how the implementations of these frameworks relate to each other, and what their performance and readability is when they are applied to a business case.

Keywords: data access frameworks, Java, Kotlin, performance analysis, code metrics

Chapter 1

Introduction

1.1 Problem statement & motivation

Almost every enterprise software application needs to retrieve, store or update persistent data. For many such software applications, data access frameworks are used in order to ease the development and increase the scalability of a software application. There are many different data access frameworks available, all exhibiting different concepts and features. However, it is not always clear in advance what framework suits best for an enterprise software application that needs to be developed.

Most programming languages provide an interface where a connection with a persistent data storage can be created, e.g. JDBC[1] in Java. However, when more sophisticated data access is needed, the use of solely a system library like JDBC might not suffice, which is likely to result in a lot of repetitive boilerplate code. Fowler described duplicate code as the *"number one in the stink parade"* of code smells. [12] Code repetition can be correlated with higher maintenance costs due to increasing code sizes and propagated defects [2, 18]. For this reason, different data access frameworks have been created at different abstraction levels in order to close the gap between persistent data storage and an object-oriented programming language like Java. This reduces boilerplate code, increases developer productivity and increases code readability. However, utilizing sophisticated data access frameworks is a double-edged sword. Some studies show that introducing additional abstraction layers can have a negative impact on performance [5] and can cause an increase of code churn in a software project [7].

Each data access framework provides different features and is used in different manners. Some frameworks assist the software developer with constructing SQL queries matching a certain schema using code generation, while others even hide the whole concept of SQL by adding an additional abstraction layer. These differences in abstraction and declarativity mean that the concepts used in these data access frameworks can be grouped together into different **data access paradigms**.

In this thesis, it is important to note the difference of the definition of the terms data access paradigm and data access framework.

- We define a **data access framework** to be a framework providing an API or interface to interact with a persistent data storage, allowing for create, read, update, and delete data access operations. Examples of data access frameworks for the Java

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.sql/java/sql/package-summary.html

platform are Hibernate[2], jOOQ [3] and JDBI[4]. Each of these frameworks provides its own set of functionalities, abstractions, and patterns to perform data access operations. These data access operations consist of data retrieval, storage, modification or deletion operations onto a persistent data storage.

- A **data access paradigm** describes similar concepts, principles and patterns among different data access frameworks, and hence represents a broader notion of how data access is performed. A data access framework can belong to one or multiple data access paradigms. This means that different data access frameworks can be grouped together into a paradigm.

Choosing the right data access framework is a critical decision during the implementation of a software architecture, because most business logic directly relates with the data access framework of choice. If the chosen framework proves to be unsuitable, significant parts of framework related code may need to be refactored, depending on the software architecture. That is why it is important that such a decision is made carefully and thoughtfully in advance, ensuring you get it right the first time. A previous thesis has been written about creating a decision support system for selecting an ORM tool and platform [28]. Another research created a decision support system for selecting a programming langauge ecosystem [11].

Because every data access framework provides exhibits features and is used in different ways, it means that it is possible to identify different paradigms of data access used by these different data access frameworks. This research project raises the question of which data access paradigms can be identified for a selection of data access frameworks in Java and Kotlin. Consequently, this research project attempts to evaluate and compare the different frameworks available in the market and try to classify these frameworks into different data access paradigms. Then, we compare the different frameworks and paradigms in two different characteristics, namely performance and code readability.

1.2 Research Goals and Questions

The general goal of this research project is to provide guidance in selecting the fit-for-purpose data access paradigm (and hence also framework) for a software project. In order to provide this guidance, we first need to identify the different data access paradigms. This gives rise to the first research question, where we will try to identify the different types of paradigms which can be identified among a selection of commonly used data access frameworks for the Java and Kotlin platforms. Formally, this results in the following research question:

RQ1. What data access paradigms can be identified among the most popular data access frameworks for Java and Kotlin?

Consequently, we want to determine the important aspects to take into account when selecting a data access paradigm for a software project. For this, we have taken two of the most important characteristics to take into account upon the decision of choosing a data access framework: the performance of the framework and the readability of the code written when implementing the framework. This introduces the following research questions.

[2] https://hibernate.org/
[3] https://www.jooq.org/
[4] https://jdbi.org/

RQ2. What is the performance of the of an implementation of a data access paradigm?

(a) What is the performance of an implementation of a data access paradigm in terms of runtime?

(b) What is the performance of an implementation of a data access paradigm in terms of processor (CPU package) energy consumption?

(c) What is the performance of an implementation of a data access paradigm in terms of memory (DRAM) energy consumption?

RQ3. How readable is the source code of an implementation that is written using the a data access paradigm?

(a) What is the Halstead metric of the implementation of a data access paradigm?

(b) How many source-lines-of-code (SLOC) does an implementation of a data access paradigm have?

1.3 Outline

This chapter introduces the problem and provides motivational arguments for this thesis. Chapter 2 discusses the relevant literature and background information for this research project. More specifically, relevant literature related to software performance and code readability are discussed here. It also demonstrates the relationship between a framework, paradigm and implementation approach. Chapter 3 gives an overview of all frameworks which have been included in this research project and discusses all of their features. Consequently, in Chapter 4, we use the features of these paradigms in order to identify the different paradigms for data access operations. We conclude this chapter by creating an overview of the different frameworks discussed in Chapter 3 and to what paradigms the implementations of these frameworks belong. Then, in Chapter 5, we introduce a test business case which will be used to conduct two experiments: one evaluating the performance and another evaluating the readability of these framework implementations. We do this by implementing the test business case with a set of data access operations for each of the identified implementation types, such that their performance and readability can be compared and analyzed. Then, we try to see whether there are significant differences among the identified data access paradigms. In Chapter 6, we discuss the results of the two performance experiments and we discuss different applicable threats to validity. Finally, in Chapter 7, we conclude the thesis by giving an answer to the research questions presented above and present a list of research directions to be explored further in the future.

Chapter 2

Background

This chapter will discuss the relevant literature of the various topics this thesis is related to. First, in Section 2.1, the ISO/IEC 25010 quality attributes related to this thesis are discussed. Then, the concept of object-relational mapping is explained in Section 2.2. We demonstrate the relationship between a data access framework, paradigm and implementation in Section 2.3. Section 2.4 discusses the literature about the performance of data access operations. Finally, Section 2.5 discusses different analytical models used to estimate the readability of source code snippets.

2.1 Quality attributes

NEN-ISO/IEC 25010:2023 [16] is a standard part of the System and software Quality Requirements and Evaluation (SQuaRE) family of standards, defining a set of characteristics and subcharacteristics of product quality models within the domain of system and software engineering. Using this standard, we can determine the various quality attributes which are relevant when comparing different data access frameworks.

- **Interaction capability**: when a framework is chosen, a team of software developers needs to interact with the data access framework of choice. If a software developer is unfamiliar with a data access framework, it is important that there are enough resources available for a software developer to learn how to use the framework. Hence, especially the subcharacteristic "learnability" is of importance here.
- **Maintainability**: about 75% of software development tasks are related to software maintenance [33]. This means that the maintainability of an implementation of a data access framework is also important. The subcharacteristics "reusability", "analysability" and "modifiability" can all be related to the readability of the source code produced when implementing a data access framework.
- **Performance efficiency**: when big quantities of data need to be processed, the performance efficiency of a data access framework is of importance. This can be measured in different degrees, like resources used or the amount of time needed to process a certain amount of data access operations.

2.2 Object-Relational Mapping

Many data access frameworks also tackle the problem of Object-Relational Mapping, or ORM for short. This concept exists within the domain of computer science and enterprise

application development for a while now. Many enterprise software applications require long-term persistent data storage. Most programming languages are able to persist data by writing files to the disk of a computer. However, using the file system of a computer for data persistence can cause data integrity issues when used concurrently and can be very inefficient. Database systems are designed for keeping track of this persistent data and allow multiple applications to access this data concurrently. However, many popular programming languages like Java, JavaScript, C++ and Python, are object-oriented. Object-oriented programming is not directly compatible with how most database systems persist their data, namely using records and tables. This causes an **impedance mismatch** [29]. These object-oriented programming languages allow for concepts like inheritance and references to other objects, while database systems store data in a different manner, having only columns and tables and relations among the records for instance. It is often difficult to represent one object as one record of the database management system; in various cases, multiple database records are required. ORM frameworks try to help addressing this problem by assisting the software developer in the translation from an object instance to a database record, and vice versa. Over the years, many programming language ecosystems have introduced ORM frameworks, like Hibernate[1] for Java, Sequelize[2] for JavaScript, and Django[3] for Python.

Most ORM frameworks also provide some way of performing data access operations. This means that ORM frameworks are a subset of data access frameworks. It is important to note the existence and origin of ORM frameworks, since many data access frameworks have been developed in order to address the impedance mismatch problem. Some of these frameworks used ORM to tackle this problem, like Hibernate. Other data access frameworks like address the impedance mismatch problem differently. For example, jOOQ[4] uses database records directly in Java instead of ORM.

2.3 Implementation approaches of frameworks

In Section 1.1, we already provided definitions for the terms **data access paradigms** and **data access frameworks**. Different data access frameworks provide various types of data access. Additionally, certain frameworks may also provide multiple approaches to performing data access operations. Hence, in this the we will provide an implementation to each of these different approaches. These implementations can belong to one or multiple paradigms. The relationship between a data access paradigm, data access framework, and framework implementation is demonstrated in Figure 2.1.

Hence, we introduce another definition for describing the different approaches in which a data access framework can be used. We define these different approaches as implementations of a data access framework. This relationship is visualized using a diagram in Figure 2.1.

Examples of usage approaches of frameworks can be given by looking at the Hibernate [5] framework for example. Hibernate's documentation has a section called "Interaction with the database" [6]. In this section, different implementation approaches for performing data access operations are demonstrated here using the Hibernate framework. Hence, a single

[1] https://hibernate.org/orm/
[2] https://sequelize.org/
[3] https://docs.djangoproject.com/en/5.0/topics/db/
[4] https://www.jooq.org/
[5] https://hibernate.org/orm/
[6] https://docs.jboss.org/hibernate/orm/6.6/introduction/html_single/Hibernate_Introduction.html#interacting

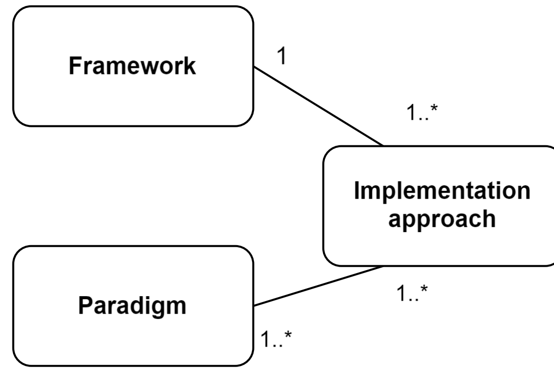

FIGURE 2.1: Diagram showing the relationships between a framework, an implementation approach and a paradigm.

framework like Hibernate can be applied using different implementation approaches. This is a recurring pattern for various data access frameworks. This means that a framework is not directly related to a paradigm, but that the implementation approach of a framework consists of certain properties which relate to a paradigm, as shown in Figure 2.1.

2.4 Performance

When creating enterprise software applications, performance is often an important factor. Application performance is often dependent on the performance of data access operations. For this reason, performance is also very important for database management systems (DBMSs). When a data access framework is used in an enterprise software application, it introduces an additional layer when performing data access operations on DBMSs. This can have an impact on performance. For some enterprise software applications, performance is very important. If an enterprise software application is not performant, it can have a direct negative impact on the overall performance of an enterprise. Therefore, it is very important to investigate the performance impact of these data access frameworks. The performance of a data access framework can be split up in two categories:

- Efficiency of the query: every data access framework eventually needs to perform a query on the persistent data storage. However, certain queries are more efficient than others. This efficiency can depend on how the data access framework is implemented or on the software developer implementing the framework.
- Runtime overhead in the JVM: when using a data access frameworks which introduces additional layers of abstraction, it is possible that additional runtime overhead is introduced.

Hence, below, we split the literature on these two topics in different subsections. To the best of our knowledge, however, no previous research has been performed on the concrete measurements of the runtime overhead of the usage of data access frameworks. A few other studies have performed overall measurements which might also include the efficiency of the query generated. Most other papers found were also specifically related to ORM frameworks, which means that not a lot of information is known about the performance impact of other types of data access frameworks.

2.4.1 Efficiency of query

Chen et al. [5] tried to detect performance anti-patterns in applications using object-relational mappings. The study looked at JPA specifically and tried to identify two common performance anti-patterns. The first anti-pattern is related to *excessive data*, where a related table is always joined upon every request while the data of this join is unused, making the join unnecessary. The second anti-pattern is called *one-by-one processing*. This means that a certain query is repeatedly performed many times because it is inside of a for-loop in Java. In many cases, this repetitive retrieval of the same data can be optimized by requesting this data in batch or by performing an optimized SQL query. The paper proposes a framework to detect these two types of anti-patterns in the source code of a software system. The performance of this framework is then evaluated by trying to find known and new performance bugs in two open-source systems and one large enterprise software system. Additionally, a prioritization of these performance problems is given such that problems having the most performance impact can be addressed first by the maintainer.

In later research, Chen et al. [7] have applied repository mining on the repositories of four systems using an ORM framework to see how often ORM related code got changed over time. This empirical study showed that code that directly interacts with ORM frameworks is more likely to change than code unrelated to ORM frameworks. Also, the researchers have posed the question of why it is the case that ORM code is more likely to change than non-ORM code. They have manually analyzed a randomized sample set of their original dataset. From this evaluation, they concluded that most ORM related code is more likely to change due to performance, compatibility or security problems. This is an indication that declarative code could cause performance issues later on, and that customization to the ORM framework is needed to address these performance issues.

Colley et al. [8] have surveyed the impact of ORM frameworks on the performance of queries. Also, they have performed an experiment using an example implementation of the Microsoft Entity Framework, demonstrating the performance impact of that framework on the query performance. They analyzed the executed SQL statements of the create, read, update and delete operations. Then, for each of these SQL statements found during runtime, a more optimal query is given by the authors of the paper if this is possible. Consequently, the two queries are entered into a SQL statement analyzer to compare the execution plans of both SQL statements. This showed that in some cases, the ORM framework does not generate the most optimal query and hence can have a negative impact on the performance of the final enterprise application. Finally, the paper concludes with a list of observed negative behaviors by the ORM, together with a suggested mitigation for each of these behaviours.

2.4.2 Overall performance

Tudose et al. [31] have compared the performance of three different ORM frameworks in Java: JPA, Hibernate and Spring Data JPA. The execution times for create, read update and delete (CRUD) operations in each of these frameworks have been measured and compared. The paper concludes with a list of pros and cons for each framework.

Bonvoisin et al. [3] have compared different configurations of state-of-the-art Java based ORM frameworks. Then, they tried to determine the energy consumption of these ORM frameworks using performance metrics. Additionally, they also included measurements of a plain JDBC implementation for comparison. The results showed that the configuration of an ORM framework has a high impact on the performance and hence also influences the

power consumption of these ORM frameworks.

Many data access frameworks utilize the Java Reflection APIs. The Oracle documentation about reflection mentions that this is a drawback, *"because reflection involves types that are dynamically resolved, certain Java Virtual Machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications."* [21]. Miao and Siek [19] have tried to address this performance issue of run-time reflection by introducing a pattern-based reflection at the statement level of the Java language. By defining an extension for the default Java language, information about classes, methods and fields can be determined at compile-time, such that the relevant reflection code can be statically generated for use at runtime. Finally, this paper applies this new pattern-based reflection in a new ORM framework called PtjORM. Miao and Siek then compared this newly proposed ORM framework to three other well-known ORM frameworks in java. In many cases, PtjORM with the optimized reflection outperformed the other ORM frameworks utilizing the default runtime reflection.

Babu et al. [1] have looked at the performance difference of Hibernate with an RDBMS and NoSQL database solution. The main goal of the research was to develop a business application independent of a database platform, allowing developers to port from RDBMS to NoSQL or vice versa whenever necessary. MySQL has been chosen as the RDBMS, and MongoDB has been selected as the NoSQL solution. The paper concludes that the performance of read and write operations of MySQL and MongoDB are almost identical. However, MongoDB did outperform MySQL in the case of update operations.

2.4.3 RAPL

In order to measure energy consumption of a machine, we have used an interface called Running Average Power Limit, or RAPL for short. This interface has been included on Intel processors since their Sandy Bridge processor series launched back in 2011. This interface allows energy metrics to be read out of Model Specific Registers from the processor. [13] Energy metrics of the processor package (PKG), dynamic random access memory (DRAM), or the graphical processing unit (GPU) can be retrieved using this interface. However, for this thesis we will only focus on the power consumption of the PKG and DRAM, since we are not utilizing the GPU explicitly.

In order to measure the energy consumption of an implementation, we used a tool called jRAPL[7]. This tool provides an easy-to-use interface for RAPL which can directly be accessed in Java. We can measure the energy consumption between two points in time by reading the values out of the Model Specific Registers. Normally speaking, the energy consumption values in these MSRs are increased over time and will wraparound to zero once the maximum value of the register has been reached. According to the documentation of jRAPL, it takes these wraparounds into account when determining the energy consumption values.

2.5 Code readability

When developing software applications, a significant amount of time of a software developer is spent on maintaining code. In order to ease the task of software maintenance, it is important that the code under maintenance is readable for software developers. One data access framework might result in more readable code than another. Hence, this factor may

[7] https://github.com/aservet1/jRAPL

also influence the decision made by a software architect regarding which data access framework to use in a enterprise software application. That is why it is important to evaluate the readability of the different data access frameworks. However, readability is a subjective concept and the notion of readability can differ per developer. Furthermore, there are many different, complementary perspectives on what it means for code to be readable [20]. In literature, many ways of quantifying readability using software metrics have been proposed. In this section, we focus on analytical models to estimate the readability of source code snippets.

Buse and Weimer [4] were the first to introduce a metric for code readability. Firstly, human annotators have been used to determine whether a code snippet is deemed readable or not. Then, they took different static code quality metrics to determine whether there is a statistical correlation among certain code features of these code snippets and their deemed readability by the human annotators. The paper showed that the average amount of lines and identifiers per line negatively impacted the readability of the code, with high predictive power. Finally, the proposed readability metric is compared with existing metrics for software quality, like code churn and defects, has been shown to be correlated.

Posnett et al. [24] continued on the work of Buse and Weimer [4] and claim to provide a more theoretically well-founded and practically usable approach to the readability measurement than the metric proposed by Buse and Weimer. The research posed the question of whether size and entropy have an influence on the readability of code. It shows that using the complexity measures defined by Halstead [15] to determine readability can outperform Buse and Weimer's readability model using the same dataset. It concludes that Buse and Weimer's readability model can be explained with a simpler model which only uses three features as proposed by Halstead: volume, lines of code and entropy. Finally, it also showed that the model proposed by Buse and Weimer is not generalizable for code snippets of bigger size.

The work of Dorn [9] investigated the influence of visual, spatial and linguistic features to the readability of source code. It shows some examples where Buse and Weimer's model misclassifies the readability of certain code snippets. Dorn showed that using these visual, spatial and linguistic features in a readability model outperforms the model proposed by Buse and Weimer in many aspects. Dorn constructed a new dataset with more human annotators then in the research of Buse and Weimer. Also, three different programming languages have been used instead of only one by Buse and Weimer.

Scalabrino et al. [27] have used more textual features in order to determine readability. Most of these additional textual features originate from natural language and linguistic theory. They then evaluated and compared these textual features with Buse and Weimer's model, Posnett's model, Dorn's model, and different permutations of those. Also, it performed this evaluation among the two existing datasets from Buse & Weimer and Dorn, and the newly created dataset. This showed that incorporating all of these features results in the best overall accuracy of 81.8%. Then, Scalabrino et al. [26] extended the study by replicating the third setion in Buse & Weimer's paper [4] by determining the correlation of code readability and warnings found by FindBugs. This showed that using the newer model, the accuracy of predicting FindBugs warnings increased.

Fakhoury et al. [10] compared and combined the models from Buse & Weimer, Posnett and Dorn, and checked whether it was possible for these models to detect readability improvements in practice. Commit messages of open source projects have been analyzed to see whether the commit was meant to perform a refactoring related to code readability. Then, they tried to verify whether the code readability model can detect an increase of readability in case the commit message suggests such an improvement of readability. The

paper concludes that the current readability models fail to capture readability improvements suggested in these commits.

2.5.1 Halstead complexity metrics

As previously mentioned by Posnett et al. [24], Buse and Weimer's model can be simplified to use only Halstead metrics. This subsection will briefly explain the metrics as defined by Maurice H. Halstead [15], by providing the most important definitions and demonstrating the theory using some code snippet examples.

Halstead metrics distinguishes tokens in a piece of code in two different types:

- **Operands** are all variables, identifiers, constants used in a certain code snippet. For example, in Java operands include `5`, `"Hello world"`, and `false`.
- **Operators** are instructions that combining one or more operands in a certain code snippet. For example, in Java, operators include `+`, `-`, and `||`, amongst many others.

Halstead metrics classify every token in a given code snippet to either the operand or operators class. Then, using these two sets, different metrics can be computed. It starts with these four basic metrics:

- η_1: unique amount of operators
- η_2: unique amount of operands
- N_1: total amount of operators
- N_2: total amount of operands

```
1 int sum(int a, int b) {
2     return a + b;
3 }
```

FIGURE 2.2: Example Java code snippet to calculate Halstead metrics for

In an article about applying Halstead metrics to a Java program, Wolle [34] classified all operators, separators and reserved keywords as operators, and all other tokens (like literals, constants and identifiers) as operands. Using these definitions on the code snippet shown in Figure 2.2, the following sets of tokens can be determined:

$$\eta = \langle \text{int}, (), \text{int}, \text{int}, \{\}, \text{return}, +, ; \rangle$$
$$N = \langle \text{sum}, \text{a}, \text{b}, \text{a}, \text{b} \rangle$$

We can turn these sets in to a table structure, where each row belongs to a certain token type, and where the columns describe the four different Halstead metrics, as shown in Table 2.1.

The following attributes as posed by Halstead can be computed using the four operator and operand measurements:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$

Token	η_1	η_2	N_1	N_2
int	1		3	
sum		1		1
a		1		2
,	1		1	
b		1		2
;	1		1	
()	1		1	
{}	1		1	
return	1		1	
+	1		1	
Total:	7	3	9	5

TABLE 2.1: Example of determining η and N Halstead metrics

- Estimated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

Hence, by utilizing the totals shown in Table 2.1, we get the following values: $\eta_1 = 7, \eta_2 = 3, N_1 = 9, N_2 = 5$. Using these values, we can for example determine the volume of the code snippet as follows: $V = 14 \times \log_2 10 \approx 46.5$.

2.5.2 Embedded complexity metrics

Previous studies have applied Halstead's complexity metrics to different types of programming languages in order to measure the complexity of code snippets. In more recent research, Halstead metrics have been applied in various contexts, including object-oriented programming languages [14] and enterprise-specific languages such as IBM RPG [30]. However, in many enterprise software applications, multiple different programming languages can be embedded inside eachother. More specifically, a general purpose programming language embeds a domain-specific language (DSL), like SQL. Figure 2.3, shows an example of this, where the host language is Java and a SQL code snippet is embedded inside a string literal.

When calculating Halstead complexity metrics for such a code snippet, the entire string literal is interpreted as a single operand. In the example of Figure 2.3, the SQL query takes up multiple lines of code and hence also introduces additional complexity to the code. However, this additional complexity is not reflected when computing the Halstead complexity metrics for this code snippet. This means that in order to obtain a more accurate measurement of complexity in a code snippet from an enterprise software application, it is important to also consider embedded languages, like DSLs. We elaborate further on this issue in Section 5.3.3.

```
1  PreparedStatement preparedStatement =
2  connection.prepareStatement(
3      """
4          SELECT i.*,
5                 n.name as legalName,
6                 d.name as documentName,
7                 d.size,
8                 dt.type_name
9          FROM individual i
10                  JOIN name n ON i.id = n.entity_id
11                  JOIN nametype nt ON n.type = nt.type
12                  JOIN document d ON d.entity_id = i.id
13                  JOIN documenttype dt ON d.type = dt.type
14         WHERE nt.type_name = 'Legal Name'
15           AND d.type = ?;""");
16   preparedStatement.setInt(1, documentType);
17   ResultSet resultSet = preparedStatement.executeQuery();
```

FIGURE 2.3: Example Java code snippet which embeds a SQL query inside a string literal.

Chapter 3

Frameworks

This chapter discusses the different frameworks which are evaluated in this thesis. Using the different properties and characteristics of the frameworks, the data access paradigms to which the different frameworks belong can be identified. First, a selection of frameworks to be included for this research project is made in Section 3.1. Consequently, the characteristics we will evaluate for each framework are discussed and explained in Section 3.2. In the following sections, these characteristics are explained for each of the different frameworks.

3.1 Framework selection

In order to complete the research goals listed above, a set of Java and Kotlin data access frameworks needs to be selected. As a starting point, we used two curated lists of popular Java and Kotlin frameworks and libraries called *"Awesome Java"*[1] and *"Awesome Kotlin"*[2]. From these lists, the entries from the categories *Database*, *ORM*, and *Platform* have been evaluated. An initial filtering has been applied, excluding all frameworks which are not related to providing data access to relational databases. This leaves us with the list of frameworks as presented in Table 3.1.
Then, we determined the popularity of these frameworks using the well-known question-and-answer website StackOverflow[3]. If a tag for the given framework exists on StackOverflow, we used this tag to find how many questions related to the framework were submitted on StackOverflow. Whenever a tag was not available for a certain framework, we searched for the name of the framework instead and added the "Java" tag. For the search terms "modality" and "requery", a lot of questions were found. However, most of these questions were not related to the framework but happened to contain the words "modality" and "requery". Because of this, we decided to exclude these two frameworks. Additionally, we have excluded "EclipseLink" because it is a Jakarta Persistence API implementation, just like "Hibernate", which is already included. Then, to limit the scope of our research, we only included the top 9 remaining frameworks based on their popularity on StackOverflow.

3.2 Evaluation

Each included framework in Table 3.1 has been analyzed and the information of each framework is given in a default format. Below, an overview is given of the various properties

[1] https://github.com/akullpp/awesome-java
[2] https://kotlin.link
[3] https://stackoverflow.com

Framework name	Platform	Category	Search term	Questions	Included
Hibernate	Java	ORM	[hibernate]	95108	✓
Spring Data	Java	Platform	[spring-data]	12033	✓
EclipseLink	Java	ORM	[eclipselink]	5091	✗
MyBatis	Java	ORM	[mybatis]	3433	✓
jOOQ	Java	Database	[jooq]	2687	✓
QueryDSL	Java	Database	[querydsl]	2009	✓
Ebean	Both	ORM	[ebean]	1107	✓
Modality	Java	Database	[java] modality [i]	500	✗
JDBI	Java	Database	[jdbi] or [jdbi3]	407	✓
Exposed	Kotlin	Database	[kotlin-exposed]	266	✓
requery	Both	Database	[java] requery[i]	170	✗
SQLDelight	Kotlin	Database	[sqldelight]	144	✓
Apache Cayenne	Java	ORM	[apache-cayenne]	133	
DBFlow	Kotlin	Databases	[dbflow]	92	
DbUtils	Java	Platform	[apache-commons-dbutils]	86	
Speedment	Java	Database	[speedment]	19	
Doma	Java	Database	doma [java][i]	9	
ktorm	Kotlin	Database	[ktorm]	9	
JINQ	Java	Database	[jinq]	5	
QueryStream	Java	Database	[java] querystream	3	

[i] Contained many unrelated results on StackOverflow

Table 3.1: List of frameworks which could be included in this research with their amount of questions on StackOverflow for the given search term. A term wrapped in squared brackets represents a tag.

which have been evaluated for each of the frameworks. Every framework evaluated is also briefly introduced with some background information about the framework.

Database management system support

For each framework, we present a list consisting of all the database management systems (DBMSs) with which the framework is able to interoperate. Because the scope of this research is limited to relational databases only, we are also only listing the support for relational DBMSs. An overview can be found in Table 3.2. It is important to note that the frameworks JDBI and MyBatis support all listed RDBMSs, as they only require a valid JDBC driver for the respective DBMS to provide support.

Implementation approaches

As mentioned in Section 2.3, some frameworks have multiple ways of performing data access operations. For example, Hibernate allows the software developer to specify their data access operations using their own query language HQL, while also providing support for the JPA Criteria API. This characteristic looks at the different implementation approaches a framework provides, and how they are different from each other.

	CUBRID	DB2	Derby	Firebird	H2	Hana	HSQL	MariaDB	MySQL	NuoDB	Oracle	PostgreSQL	SQL Anywhere	SQLite	SQL Server	Sybase	Teradata	TiDB	Trino
Ebean		✓			✓	✓			✓	✓	✓	✓	✓	✓	✓				✓
Exposed					✓			✓	✓		✓	✓		✓	✓				
Hibernate		✓	✓		✓		✓	✓	✓		✓	✓			✓			✓	
JDBI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
jOOQ			✓	✓	✓		✓	✓	✓			✓		✓					✓
MyBatis	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
QueryDSL	✓	✓	✓	✓	✓		✓		✓		✓	✓		✓	✓				
Spring Data		✓					✓	✓	✓		✓	✓			✓		✓		
SQLDelight		✓			✓		✓		✓		✓	✓		✓					

TABLE 3.2: Table of supported relational database management systems per data access framework.

Type safety

Some frameworks provide type checks against the database schema or the POJO defintions of some of the tables. In that case, the type safety provided by the framework is briefly discussed.

Object mapping

Most frameworks support some type of object mapping. When performing a query using JDBC, the resulting records of the query are returned using a *ResultSet*. This *ResultSet* is often difficult to utilize directly in other parts of the business logic, without resorting to a lot of boilerplate code. That is the main reason why most data access frameworks provide some support for mapping these result sets towards Java objects. In this section, the type of object mapping supported by the given framework is briefly explained.

Caching

In many cases, the performance of data access operations can be optimized by minimizing the amount of redundant data accesses. [6] One way of reducing the amount of accesses to a database is by introducing a caching layer. Some of the frameworks under review provide this caching mechanism in the data access layer. This section mentions the types of caching supported by the framework and whether the documentation mentions if integration with third-party caching solutions is also supported. Most frameworks distinguish two types of caches: a first level cache and a second level cache. A first level cache is session-specific and automatically stores data during a transaction, clearing once the session ends. A second level cache is shared across sessions, providing broader, persistent caching to reduce database hits for frequently accessed data.

Stored procedures

A stored procedure is a set of SQL statements that can be stored and executed on a database server. Some database operations can benefit from a performance improvement when executed as a stored procedure on the database. This section discusses how stored procedures or native function calls are supported in the framework.

3.3 Ebean

Ebean is an ORM framework created by Rob Bygrave in 2012. It is available for both Java and Kotlin.

Implementation approaches

Ebean provides three different abstraction levels for performing data access operations.

- **ORM Queries**. This implementation approach uses a generated query class, which contains metadata in order to provide a typesafe manner of performing data access. This means that no SQL needs to be written specifically. The documentation claims that 85% of the queries in recent applications are "pure" ORM queries. If custom functions are needed which the DSL does not provide, the *select* or *where* clause can still be added as SQL using the DSL.
- **DTO Queries**. Using this implementation approach, the query is specified as SQL but the result records are automatically mapped to a data transfer object (DTO).
- **SqlQuery**. Using this implementation approach, the query is specified as SQL and the results are represented as *SqlRow* objects, which has a similar approach and API as the default Java *ResultSet*. However, the *SqlRow* offers methods to specify a limit and offset to result list, which then automatically gets applied in the appropriate SQL dialect.

Type safety

When using the ORM Queries implementation approach, queries are typesafe because the metadata of a table contained in the query objects. This means that in the case of a tye error, an error will be raised during compilation. In the other DTO and SqlQuery approaches, the types are not checked and hence no type safety is guaranteed.

Object mapping

Ebean uses JPA anootations for mapping the database columns to Java objects. The DTO Queries approach allows columns from a result set to be mapped to a DTO bean, where the names of the columns need to match the getter/setter methods in the DTO bean.

Caching

Ebean provides support for a L2-cache[4]. This cache contains two types of data: bean caches and query caches. A bean cache consists of entity beans mapped from their unique identifier to the entity bean. This means that queries using a "find-by-id" or "find-by-natural-key" can be optimized using these bean caches. The query cache holds a list of executed queries and their results. When data in these queries gets changed, the cache automatically gets invalidated. Both cache types can be enabled using an annotation in the entity class.

Stored procedures

Stored procedures can be called using the SQL Query implementation approach by writing the SQL needed for the stored procedure.

[4] https://ebean.io/docs/features/l2cache/

3.4 Exposed

Exposed is a Kotlin framework introduced by JetBrains in 2013. It provides a domain specific langauge to specify structures of a database as Kotlin objects. Using this DSL, SQL queries can also be specified inside Kotlin, abstracting the concept of SQL altogether. Another submodule also adds support for data access using data access objects (DAOs).

To date, non-relational database systems are not supported by the Exposed framework yet. However, it is likely that this feature will be implemented in the future, since there is a separate module in the repository called `exposed-jdbc`, which is the *"transport level implementation based on Java JDBC API"*. This suggests that the Exposed framework can be extended with an additional transport layer for non-relational database systems.

Implementation approaches

Exposed provides two different implementation approaches for specifying data access operations: a typesafe SQL wrapping domain specific language, and a DAO domain specific language.

- **SQL wrapping DSL**. This implementation approach in Exposed is *"similar to actual SQL statements, but with the type safety that Kotlin offers"*. First, the Table type needs to be defined in Kotlin, as shown in Figure 3.1. Then, inserts and queries can be performed on this table definition as shown in Figure 3.2.

```
1 object Blogs : Table() {
2     val id: Column<UUID> = uuid("id")
3     val title: Column<String> = varchar("name", length = 50)
4     val content: Column<String> = varchar("content", length = 2000)
5
6     override val primaryKey = PrimaryKey(id)
7 }
```

FIGURE 3.1: Example of a Table definition in Exposed

- **Data Access Objects**. This approach is *"is similar to ORM frameworks like Hibernate with a Kotlin-specific API"*. For the usage of this syntax, the additional `exposed-dao` module is required. In addition to the table definition as shown in Figure 3.1, a Kotlin class for the table also needs to be defined. An example for this is shown in Figure 3.3, together with what a lookup using this DAO object would look like. The software developer can define additional functions specific to that class. In this example, the `toString` function has been overriden.

Type safety

Exposed provides type safety due to the way their DSL is implemented. Types are defined as an object for a table, as shown in Figure 3.1. This enables the DSL to provide type checks upon compilation.

Object mapping

Kotlin allows the database schema to be defined in-code using Kotlin objects. An example of this is shown in Figure 3.1.

```
transaction {

    Blogs.insert {
        it[id] = UUID.randomUUID()
        it[title] = "My first blog"
        it[content] = "Hello world, this is my first blog!"
    } get Blogs.id

    Blogs.insert {
        it[id] = UUID.randomUUID();
        it[title] = "Second blog"
        it[content] = "This is already my second blog, wow."
    }

    for(blog in Blogs.selectAll()) {
        System.out.printf("%s: %s\n", blog[Blogs.title],
            blog[Blogs.content]);
    }

    for(blog in Blogs.selectAll().where {
        Blogs.title like "%first%"
    }) {
        System.out.println("The following blogs contain 'first' in their
            title:")
        System.out.printf("%s: %s\n", blog[Blogs.title],
            blog[Blogs.content]);
    }
}
```

FIGURE 3.2: Example of the Exposed SQL DSL

Caching

When investigating at the source code of Exposed, there seems to be a transactional level entity cache[5]. However, no real first-tier or second-tier caching mechanisms have been developed for Exposed yet. This means such caching functionality needs to be addressed and implemented by the software developer themselves.

Stored procedures

When inside a transaction, it is possible to call stored procedures using the `exec()` function inside the transaction context. This performs direct SQL on the transaction. It is also possible to define custom SQL functions[6] to be used inside the DSL. However, this requires a return type, and hence cannot directly be used with stored procedures.

[5] https://github.com/JetBrains/Exposed/blob/main/exposed-dao/src/main/kotlin/org/jetbrains/exposed/dao/EntityCache.kt

[6] https://jetbrains.github.io/Exposed/sql-functions.html#custom-functions

```kotlin
class Blog(id: EntityID<UUID>) : Entity<UUID>(id) {
    companion object : EntityClass<UUID, Blog>(Blogs)
    var title by Blogs.title
    var content by Blogs.content
    override fun toString(): String {
        return String.format("%s: %s\n", title, content)
    }
}

transaction {
    val newBlog = Blog.new {
        title = "My first blog"
        content = "Hello world, this is my first blog!"
    }

    val secondBlog = Blog.new {
        title = "Second blog"
        content = "This is already my second blog, wow."
    }

    for(blog in Blog.all()) {
        println(blog)
    }

    for(blog in Blog.find(Blogs.title like "%first%")) {
        println(blog)
    }
}
```

FIGURE 3.3: Example of the Exposed DAO API

3.5 Hibernate

Hibernate ORM [7] provides a Object/Relational-mapping between the Java memory model and relational databases. It has been created in 2001 by Gavin King as an attempt to offer better persistence features in Java than provided by EJB2, which was the most commonly used persistence framework back in the day. The developers have later been hired by JBoss to continue the development of the Hibernate framework. In 2006, JBoss has been acquired by RedHat. In April 2024, the Hibernate project has announced that it will be moved to the Commonhaus Foundation[8].

For some database systems, there are also additional dialects available to be used within Hibernate. It is also possible to define your own custom dialect using their API.

Implementation approaches

Hibernate provides three different implementation approaches for performing data access operations.

[7] https://hibernate.org/

[8] https://www.commonhaus.org/activity/123.html

- **Hibernate Query Language**[9]. This customized query language is a superset of the Jakarta Persistance Query Language (JPQL) and is able to abstract modern langauge features from several SQL dialects. This allows a query to be written once in HQL, which is then supported for multiple SQL dialects. Queries for HQL are written as a String.
- **JPA Criteria API**. This way of querying provides an implementation of the JPA Criteria specification. Using the Hibernate implementation, queries are written in a typesafe manner.
- **Native query**. These queries are executed directly on the database and hence are not database agnostic.

Type safety

Hibernate has a metamodel generator. This metamodel generator generates classes for each of the table instances of a given database schema, giving the software developer easy access to information about the database schema used. This is helpful when writing queries using the JPA Criteria API, allowing queries to be written in a typesafe manner. The metamodel generator also ensures that whenever a name or type of a column in the database changes, this is noticed for all occurrences at compile time.

Named query annotations are also typechecked, whilst the queries of the HQL and Native SQL implementation approaches are written as strings, and hence are not type checked.

Object mapping

Hibernate describes itself as an ORM framework, and hence it also has great support for mapping the results of a query to a Java object. The objects can be defined in Java using the JPA annotations as specified in the JPA specification.

Caching

The documentation of Hibernate mentions two different kinds of caching:

- **First level cache**. This level caches the persistence context. This means that if within the same session the same persistence context is required twice, it is only retrieved once and reused the second time.
- **Second level cache**. This allows certain persistence objects to be reused over multiple sessions. In order to prevent any violation of the ACID-properties[10], this second level cache is only eligible for specific persistence tables. These tables can be marked as eligible for second level cache using the `@Cache`-annotation.

Stored procedures

Hibernate has support for stored procedures, as it is also specified in the JPA specification[11]. However, in the Hibernate implementation, the name of the method is slightly

[9] https://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html

[10] ACID is a common abbreviation used in the context of datbase managemnt system, and stands for the properties atomicity, consistency, isolation and durability.

[11] https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#createStoredProcedureQuery-java.lang.String-

different: here it is called `createStoredProcedureCall`, while in JPA it is called `createStoredProcedureQuery`. However, the functionality is similar.

3.6 JDBI

JDBI is an open-source Java library created by Brian McCallister in 2004. It was to *"provide convenient tabular data access in Java"* [12]. Currently, it is hosted on GitHub and is still maintained by the open-source community. It is is currently at its third major release and now also has support for other JVM languages like Kotlin, Clojure and Scala.

Implementation approaches

Using JDBI, two different APIs are provided for performing queries against a database.

- **Fluent API**. This API uses a builder pattern to construct a SQL query. First, the SQL query is given as a string without the parameters. Then, dynamic parameters can be bound to the query, together with any additional query settings. Consequently, the results of the query can be retrieved and processed.
- **Declarative API**. This API utilizes annotations above interface methods to specify queries and is provided as an additional module to JDBI, called SQL Object extension[13]. Using this API, the entire query still needs to be provided inside of the annotation. However, one of the main differences compared to the Fluent API is that binding of parameters can be performed implicitly by adding them as parameters to the method with the correct type and name. Also, the `@Bind` parameter can be used for explicit parameter binding.

Type safety

JDBI has no assumptions or information on the database schema at compile time. This means that all SQL queries provided to JDBI are not checked against a database schema for type correctness upon compilation. This means that JDBI cannot provide any type safety guarantees and that this responsibility is given to the software developer.

Object mapping

JDBI clearly describes in the introduction section of their documentation [14] that it is not an ORM or complete database management framework. They do provide some ORM-like functionality, but it is in no sense close to what other frameworks have to offer. `Mapper`[15] classes can be used to create a mapping from a `java.sql.ResultSet` to another object type. However, all these mappers need to be created by the software developer manually.

Caching

Only the parsing of certain statements are cached by default according to their documentation[16]. No other caching mechanisms are provided by default, which means that the software developer should either develop their own caching mechanism if desired, or should

[12] http://kasparov.skife.org/jdbi/
[13] https://jdbi.org/#sql-objects
[14] https://jdbi.org/#_getting_started
[15] https://jdbi.org/#_mappers
[16] https://jdbi.org/#_statement_caching

use a third-party caching framework together with JDBI. The JDBI documentation does not mention any integration examples of caching mechanisms.

Stored procedures

JDBI does have support for stored procedure calls[17] by defining the output parameters of the procedure call. These output parameters can then be used in Java again.

3.7 jOOQ

jOOQ stands for "Java Object Oriented Querying" and is developed by a German IT company called Data Geekery. It attempts to stay as close to SQL as possible by providing a DSL which is very SQL-like. jOOQ has four different editions, of which the free tier is open-source. Support for older LTS versions of Java is only provided for the three paid editions, together with additional support for commercial database management systems like SQL Server or Oracle. These are the four editions available:

- Open Source
- Express
- Professional
- Enterprise

The jOOQ documentation that there are many different ways of integrating jOOQ within a software project. It can be used next to the Hibernate framework for example.

Implementation approaches

jOOQ allows queries to be constructed using their SQL builder DSL. The DSL is very similar to actual SQL code. Figure 3.4 shows an example usage of the query builder. The software developer needs to create a `Connection` themselves using the JDBC API.

```
1  try(Connection conn = DriverManager.getConnection(JDBC_URL)) {
2      DSLContext create = DSL.using(conn, SQLDialect.SQLITE);
3
4      Result<Record> result = create.select().from(BLOGS)
5              .where(BLOGS.TITLE.like("%second%"))
6              .fetch();
7
8      System.out.println(result);
9  } catch (Exception e) {
10     e.printStackTrace();
11 }
```

FIGURE 3.4: Example of the SQL builder DSL of jOOQ

[17] https://jdbi.org/#_stored_procedure_calls

Type safety

jOOQ tries to assure type safety by using code generation. The code generator can be connected to your existing database schema, of which then a metamodel will be generated in Java. This metamodel can then be used while constructing queries. While constructing the queries, the types are then also immediately checked and verified. This can also be seen in the code snippet in Figure 3.4. On line 5, the `title` field is not referenced using a string literal, but utilizes the generated metamodel class of the Blogs table. This means that if the name or type of the field were to change, this is immediately noticed upon compile time.

Object mapping

jOOQ has support for POJO mapping back into records[18], and also has a code generator which will generate these POJOs from your database schema.

Associationns among entities can also be defined when utilizing JPA annotations for the POJo objects. However, associations in jOOQ always need to be loaded explicitly in code, and are never eagerly fetched automatically[19].

Caching

The jOOQ documentation[20] mentions that a first or second level cache is not needed because this would be the opposite of the idea of jOOQ, namely that every interaction towards your database is performed using only queries. There are no existing entity projections in memory, hence there is also no need to cache such information. If there is the need for a cache in order to optimize performance, an existing caching solution of choice can be integrated manually above the querying layer.

Stored procedures

jOOQ has support for stored procedures according to their documentation[21]. For stored procedures and functions, an `org.jooq.Routine` object is also generated using their code generator. These routines can then be called in-code, and required parameters have getters and setters, which also ensures that stored procedures can be used in a type-safe manner.

3.8 MyBatis

MyBatis is a free, open-source persistence framework is a fork of the iBATIS project. The iBATIS project was created in 2001 by Clinton Begin and initially focused on the development of cryptographic software solutions. Later, the development team focussed more on developing data access object solutions, allowing for automated mappings between SQL databases and objects in the programming languages Java, .NET and Ruby on Rails. MyBatis provides a similar functionality but is only focused on Java. It allows objects to be mapped to SQL statements using XML-defined mappings. In later releases, support for annotations have also been added to the framework.[22]

[18] https://www.jooq.org/doc/3.19/manual/sql-execution/fetching/pojos/
[19] https://www.jooq.org/doc/3.19/manual/coming-from-jpa/from-jpa-eager-lazy/
[20] https://www.jooq.org/doc/latest/manual/coming-from-jpa/from-jpa-caches/
[21] https://www.jooq.org/doc/latest/manual/sql-execution/stored-procedures/
[22] https://blog.mybatis.org/p/about.html

Implementation approaches

MyBatis has two different implementation approaches:

- **XML mappings**. In separate XML files, SQL queries can be defined. These queries have a namespace and unique ID, so that they can be called from the MyBatis API inside the Java project. Also, the return type of the query is given. MyBatis will try to automatically map the resulting SQL columns to the given result type. It also has support for automatically inserting parameters inside a SQL query using a injection-safe manner. The XML format also has support for so called "Dynamic SQL". Additional XML can be inserted inside of the SQL query, allowing for conditional additions to the SQL query.

```
1 <mapper namespace="nl.utwente.BlogMapper">
2       <select id="selectBlog" resultType="nl.utwente.Blog">
3       select * from Blog where id = #{id} WHERE state = 'ACTIVE'
4             <if test="title != null">
5                   AND title like #{title}
6          </if>
7    </select>
8 </mapper>
```

FIGURE 3.5: MyBatis XML mapping example

In Figure 3.5, the `LIKE` clause on line 5 is only added when the value of title is also given upon runtime.

- **Annotations**. The SQL queries can also be added above interface methods using annotations. Here, the name, return type and parameters of a query are defined in the interface method instead of in an XML mapping. Figure 3.6 shows an example of an equivalent mapping given in Figure 3.5, but then using an annotation.

```
1 @Select("SELECT * FROM Blog WHERE id = #{id}")
2 Blog selectBlog(int id);
```

FIGURE 3.6: MyBatis annotation example

Dynamic SQL is also possible using the SQL Builder API. First, a SQL query is constructed and returned as a string. This is then directly plugged into an interface method. An example of this is shown in Figure 3.7.

Type safety

MyBatis does not add any type safety to their querying methods. There are no compile time checks which will check whether the queries given to the database have the correct type. Queries are just executed as given.

Object mapping

MyBatis does provide mapping from database records towards Java beans or POJO classes. It can also map OneToMany-relationships, but requires the software developer to manually

```java
public interface BlogDAO {

        @SelectProvider(type = SqlBuilder.class, method="selectBlogSql")
        Blog selectBlog(int id);

}

public class SqlBuilder {
        public static String selectBlogSql() {
            return new SQL() {{
                SELECT("*");
                FROM("Blog");
                WHERE("id = ${id}");
            }}.toString();
        }
}
```

FIGURE 3.7: Example of query builder

map these using a ResultMap XML definition, which is then bound to the result of a query.

Caching

The MyBatis documentation does mention cache support[23]. By adding a `<cache />` tag to the XML mapping file, all results from select statements from the mapping file will be cached. Insert, update and deletes will invalidate the cache.

It also has support for implementing your own cache or using third party caching solutions.

Stored procedures

When a stored procedure needs to be called, this can be inserted inside the XML tags or annotations natively, since the queries are executed as specified in the XML-mapping or annotation.

3.9 QueryDSL

QueryDSL is a domain-specific language in Java written by Timo Westkämper in 2007. Its main goal was to provide a type-safe manner for constructing Hibernate Query Language (HQL) queries[24]. Later, support for other back-ends than JPA and Hibernate have also been added to the framework, like SQL, JDO and MongoDB.

Implementation approaches

QueryDSL has two approaches of using their DSL.

- **QueryDSL JPA**. This approach is based upon JPA annotations and provides a DSL based approach for writing JPQL queries.

[23] https://mybatis.org/mybatis-3/sqlmap-xml.html#cache
[24] http://querydsl.com/static/querydsl/1.0.0/reference/html/ch01.html

- **QueryDSL SQL**. This approach provides a DSL for writing SQL queries.

Both approaches of QueryDSL provide a code generation step in Maven, which will generate metamodel classes for the schema of use. Then, using a query factory, database queries can be constructed. The code snippet in Figure 3.8 shows an example of what an insert and select statement look like in QueryDSL using their SQL approach.

```
1  SQLQueryFactory queryFactory = createQueryFactory();
2  QBlogs qBlogs = QBlogs.Blogs;
3
4  queryFactory.insert(qBlogs)
5          .columns(qBlogs.title, qBlogs.content, qBlogs.id)
6          .values("Hello, world!", "This is my first blog.",
               UUID.randomUUID())
7          .execute();
8
9  List<Tuple> tuples =
       queryFactory.query().select(qBlogs.all()).from(qBlogs).fetch();
10
11 for(Tuple t : tuples) {
12     String title = t.get(qBlogs.title);
13     String content = t.get(qBlogs.content);
14     System.out.println(title + ": " + content);
15 }
```

FIGURE 3.8: QueryDSL insert and select example

Type safety

QueryDSL has a metamodel generator which can be used for ensuring type safety in queries written using QueryDSL. Instead of referencing the columns and table names using string literals, the generated metamodels can be used, which will also ensure constructed query does not contain any type errors. This is ensured using the Java compiler, because compilation will fail in case of a type mismatch.

Object mapping

Depending on the type of back-end, QueryDSL can perform some type of object mapping. In the case of using a SQL back-end, no mapping happens by default and a `Tuple` type is returned.

When using JPA as back-end, fetched results can automatically be mapped to the JPA entities.

Caching

QueryDSL by default applies no caching and leaves this up to the user to implement. When using a JPA back-end, it is possible that the JPA implementation offers caching possibilities. However, this is dependent on the JPA implementation of choice.

Stored procedures

Using QueryDSL, it is not possible to directly call stored procedures or other user defined database functions. The best way to solve this is by re-using the `Connection` of the `QueryFactory`, and then creating a `PreparedStatement` yourself with the correct query.

3.10 Spring Data

Spring Data [25] is a Spring-based data access framework. It supports both relational and NoSQL database solutions, as well as some in-memory solutions. One of the subprojects of Spring Data is called Spring Data JPA[26], which eases the use of the Java Persistence API within the Spring framework. Using Spring Data JPA, repository interfaces can be created of which SQL queries will be generated at runtime by the framework. Under the hood, it also utilizes parts of the Hibernate ORM framework.

According to the list of supported database types in the Spring Data JDBC repository [27], the following relational database systems are supported:

- HSQL
- MySQL
- MariaDB
- Postgres
- Microsoft SQL Server
- Oracle
- DB2

Implementation approaches

Spring Data provides three different implementation approaches.

- **Query methods**. Queries are extracted from method interface names, with `findByEmailAddressAndLastName` automatically generating the correct SQL query. Parameters of this interface method are then also used for the generating query.
- **JPA Named Queries**. JPA queries can be specified in an XML file and referenced above interface methods using `@NamedQuery`.
- **Query annotation**. This annotation is also given above interface methods, but then contains the SQL query to be executed as a string argument in the annotation. This also has support for so-called "native queries", which disables query rewriting for the dialect as selected in the settings of Spring Data, and also has no support for pagination or dynamic sorting. However, a ‘countQuery‘ property can also be supplied which gets the count of total records for the native query, which will then still support pagination.

[25] https://spring.io/projects/spring-data
[26] https://spring.io/projects/spring-data-jpa
[27] https://github.com/spring-projects/spring-data-relational/tree/main/spring-data-jdbc

Object mapping

Spring uses JPA annotations to specify the mapping from database records to Java objects. Also, the relationships between objects can be specified using the JPA annotations, allowing the software developer to automatically request the related entities of a certain object.

Caching

In Spring Data JPA, cache solutions are not directly provided. However, there are some extensions which can provide caching functionality, like Spring's `CacheManager`.

Stored procedures

Spring Data JPA has support for stored procedures. A stored procedure can be mapped to an interface method using the `Procedure` annotation. In case there is only a single output parameter, the type of the method is equal to the type of that single output parameter. In case that there are multiple output parameters, these output parameters are returned as a map.

3.11 SQLDelight

SQLDelight is a Kotlin framework developed by CashApp in 2016. It uses an existing database schema to generate typesafe API calls for any queries that need to be executed. These queries are specified outside of the regular Kotlin files.

Implementation approaches

In SQLDelight, SQL queries are specified in separate `.sq` files. Inside these files, multiple SQL statements can be specified with a name. Also, an initial schema and seeding can be provided. Figure 3.9 shows an example of what such a `.sq` file looks like.

```
1 selectAll:
2 SELECT * FROM Blogs;
3
4 selectTitleLike:
5 SELECT * FROM Blogs WHERE title LIKE ?;
```

FIGURE 3.9: An example of an `Blog.sq` file in SQLDelight.

Then, SQLDelight will generate an object called `BlogQueries`, which contains typesafe functions for the specified SQL statements. These can then be used directly in Kotlin. Figure 3.10 shows an example of this. The code generator of SQLDelight recognized that the input type of the parameter should be a string, and hence the type of the paramter in the generated `selectTitleLike` function also has the type string.

Type safety

As shown in Figure 3.10, the generated functions have the correct types. Hence, SQLDelight does guarantee type safety of the written queries.

```
val blogs = blogQueries.selectAll().executeAsList();

for(blog in blogs) {
    println(blog)
}

val blogsWithFilter =
    blogQueries.selectTitleLike("%world%").executeAsList();

for(blog in blogsWithFilter) {
    println(blog)
}
```

FIGURE 3.10: Example usage of the generated `BlogQueries` object.

Object mapping

By default, a Kotlin data class is generated for each table with all columns of that table. If all columns of one table are given in the select clause, this data class is automatically the return type of the query. If there is a different select clause, a Kotlin data object with the name of the query is generated, together with all columns specified in the select clause. This means that the result records are directly mapped to a Kotlin data class. However, associations are not shown here.

Caching

The documentation mentions nothing about any caching mechanisms implemented in the framework. This means that this responsibility is given to the software developer implementing the framework.

Stored procedures

SQLDelight does not support stored procedures using their `.sq` files. Raw queries can be used instead in order to still call stored procedures.

Chapter 4

Paradigms

In Chapter 3, we have reviewed a list of frameworks for the Java and Kotlin platforms. Using the information of each of these frameworks, we will try to classify and group their implementation approaches into different types of paradigms. For this approach, we have chosen to use two different characteristics in which we will try to order the different implementation approaches, namely *declarativeness* and *abstraction*. Then, we will try to introduce the concepts which we will use in order to divide the different implementation approaches into paradigms. Finally, we present an overview of the identified data access paradigms.

4.1 Declarativeness

The characteristic declarativeness in the context of data access frameworks describes the manner in which the data access is specified.

- **Imperative**: imperative programming means that a program is made up from a clearly-defined sequence of instructions to a computer. In the context of data access frameworks, this means that the software developer has full control of the SQL query being executed in order to perform a data access operation, for example using JDBC.
- **Declarative**: declarative programming means that the goal to be achieved has been specified, but the steps of how this goal should be achieved are not specified. In the context of data access frameworks, this means that the desired result is specified without the need of writing the full SQL query. For example, Spring Data allows interface methods to be specified, where the SQL query is generated based on the name that is given to a method.

The degree to which a data access framework aligns with either "declarative" or "imperative" labels can vary, sometimes falling between the two or exhibiting varying degrees of both concepts. Also, a single data access framework may incorporate both declarative and imperative approaches to specifying data access operations.

4.2 Abstraction

The characteristic *abstraction* in the context of data access frameworks describes the layers of abstraction provided for the software developer using the data access framework. Data access operations for relational database management systems in Java are always performed

using a JDBC connection. Most data access paradigms try to provide some level of abstraction for the software developer. This characteristic describes the degree of abstraction a certain framework provides. The closer it is to plain JDBC, the less abstraction is used.

4.3 Relation

The relation between *declarativeness* and *abstraction* is not entirely orthogonal. Intuitively speaking, it makes sense to say that the more declarative a certain syntax is, the more abstraction it gives as well. In order to provide the software developer with a more declarative syntax, more abstraction layers need to be build upon the imperative programming language in order to make it more declarative. However, the amounts of abstraction and declarativeness can differ among the different frameworks. Hence, in this research, we have decided to create a two-dimensional grid in order to classify the different types of frameworks. An example of such a grid is given in Figure 4.1.

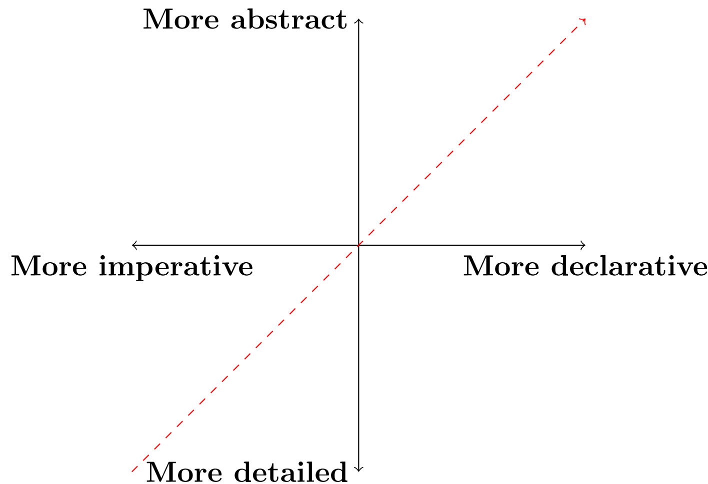

FIGURE 4.1: Example grid of specificity on the horizontal axis and abstraction on the vertical axes. The red dotted line is the suggested relationship between specificty and abstraction.

4.4 Implementations of the frameworks

Despite different frameworks providing different types of data access, an individual data access framework can also provide different types of data access. We define these different ways of utilizing a single framework as an implementation approach. These different approaches have been mentioned in the *Implementation approaches* part of Chapter 3. These different implementation approaches of frameworks can also have different levels of specificity and abstraction, hence we will classify these separately as well. In Table 4.1, the different Frameworks and their implementation approaches can be found.

For each of the approaches shown in Table 4.1, we can put them into the grid shown in Figure 4.1. The relative distribution of the approaches of the frameworks can be seen in Figure 4.2.

Framework	Implementation approach	Abbreviation
Ebeans	ORM	EB-ORM
	DTO	EB-DTO
	SQL	EB-SQL
Exposed	DAO API	E-DAO
	SQL DSL	E-DSL
Hibernate	Native query	H-NQ
	JPA criteria	H-JPAC
	HQL	H-HQL
JDBI	Fluent	JDBI-F
	Declarative	JBDI-D
jOOQ		jOOQ
MyBatis	XML mappings	MB-XML
	Annotations	MB-A
QueryDSL	JPA	QDSL-JPA
	SQL	QDSL-SQL
Spring Data	Named queries	SD-NQ
	Annotations	SD-A
	Query methods	SD-QM
SQLDelight		SQLD

TABLE 4.1: List of frameworks with their different implementation approaches and their abbreviations.

4.5 Concepts

After discussing different properties for each of the frameworks in Chapter 3, we can define different feature sets for each of the frameworks. This section discusses the different definitions for each of these feature sets. Then, for each of the different implementation approaches, we can specify which implementation approach utilizes the different concepts.

4.5.1 Specification of data access operations

The most important part of a data access framework is how the data access is specified. Using our findings discussed in Chapter 3, we have been able to identify the following specifications:

- **String manipulation**. SQL queries are written as a string and the framework allows parameters to be inserted easily and securely.
- **Configuration files**. The SQL queries to be performed are stored in an external configuration file, separate from the rest of the code.
- **Metadata**. The data access operation to be executed is specified in the metadata of a programming language, for example using annotations and interface methods. The parameters of the interface method are used as parameters for the query to be executed. The result is automatically mapped to an object of the specified return type.
- **DSL**. Data access operations are specified using a domain specific language specified as an API in Java or Kotlin. The APIs used are typesafe and are based on the domain model.

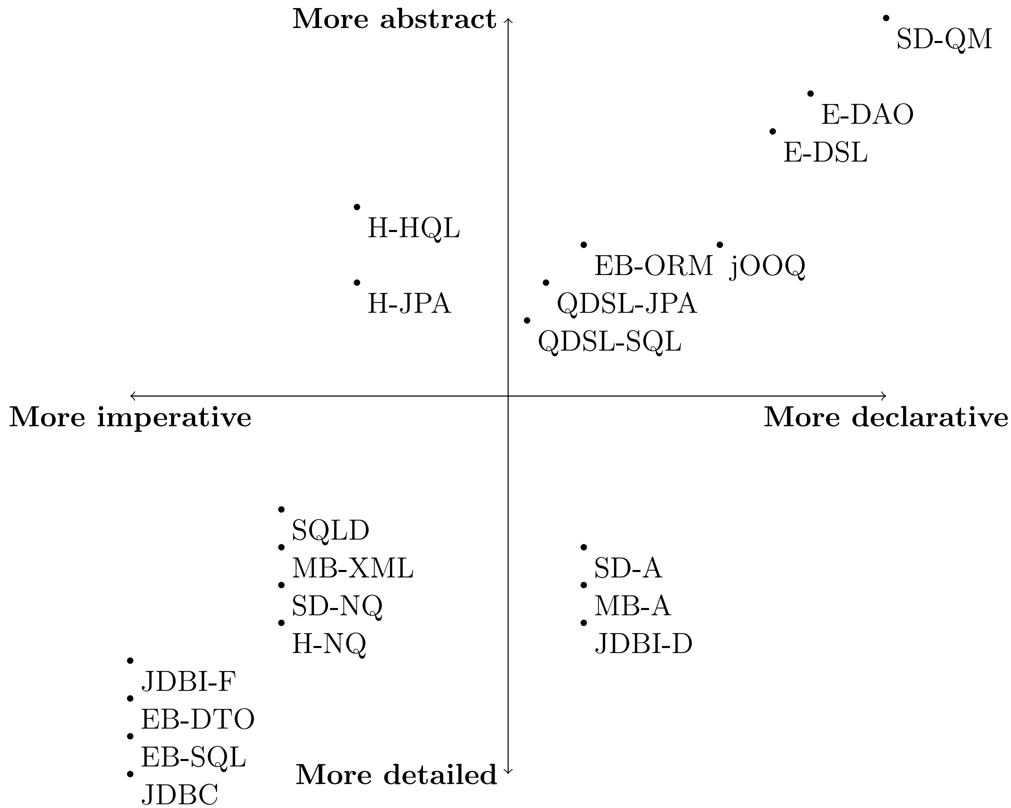

FIGURE 4.2: A relative division of the implementation approaches on a two-dimensional grid, with the vertical axis describing the abstraction and the horizontal axis describing the declarativeness.

- **Convention**. The data access operation to be executed is specified inside the syntax of the programming language itself, for example by using a specific naming convention for the names of methods. Based upon how these methods are named, the query to be executed is determined.

4.5.2 Database agnosticism

A data access framework is **database agnostic** if the database management system used can be replaced with another database management system without the need to change any of the data access operations. For example, query languages like the Jakarta Persistence Query Language (JPQL) or the Hibernate Query Language (HQL) allow the software developer to write queries which can be translated to multiple SQL dialects. We have noticed that certain groups of data access frameworks support database agnosticism, while others do not provide this feature. Database agnosticism can be an important requirement during the selection process of a data access framework, because it gives the software architect more flexibility for the choice of database management system. Hence, we call the frameworks of which the data access operations are not written for a specific database management system **database agnostic**, and the frameworks were data access operations are written specifically for one database management system **database specific**.

4.5.3 Domain metamodel

We have found that there are two different ways in which a data access framework can specify its domain metamodel:

- **Generated metamodel based on database schema**. The build tool needs access to a database containing the schema of the domain model. This build tool then generates the model of the domain in Java or Kotlin classes, which can then be used to specify type-safe data access operations which exactly match the database schema. When changes in the schema are applied, they are directly type checked by the Java compiler.

- **Manually specified metamodel**. The software developer needs to specify the domain metamodel themselves in the Java code and needs to ensure that this model matches the schema of the database. Sometimes, the framework can also generate a database schema based upon the manually specified domain model. This is called a code-first approach.

4.6 Paradigm overview

All concepts mentioned in Section 4.5 can be evaluated for each of the framework implementation approaches mentioned in Table 4.1. An overview of this is given in Table 4.2.

Implementation approach	Specification	Database agnosticism	Domain metamodel
Ebeans ORM	DSL	Agnostic	Generated
Ebeans DTO	String manipulation	Specific	Manual
Ebeans SQL	String manipulation	Specific	Manual
Exposed DAO	DSL	Agnostic	Manual
Exposed DSL	DSL	Agnostic	Manual
Hibernate Native Query	Metadata	Specific	Manual
Hibernate JPA Criteria	Metadata	Agnostic	Manual
Hibernate HQL	Metadata	Agnostic	Manual
JDBC	String manipulation	Specific	Manual
JDBI Fluent	String manipulation	Specific	Manual
JDBI Declarative	Metadata	Specific	Manual
jOOQ	DSL	Agnostic	Generated
MyBatis XML	Configuration files	Specific	Manual
MyBatis Annotations	Metadata	Specific	Manual
QueryDSL JPA	DSL	Agnostic	Generated
QueryDSL SQL	DSL	Agnostic	Generated
Spring Data Named Queries	Configuration files	Specific	Manual
Spring Data Annotations	Metadata	Agnostic	Manual
Spring Data Query Methods	Convention	Agnostic	Manual
SQLDelight	Configuration files	Specific	Generated

TABLE 4.2: The three different concepts for each of the different implementation approaches.

Using a similar approach as in Van Roy's paper [32] explaining the different programming paradigms, we can construct a visual overview of each of the different paradigms

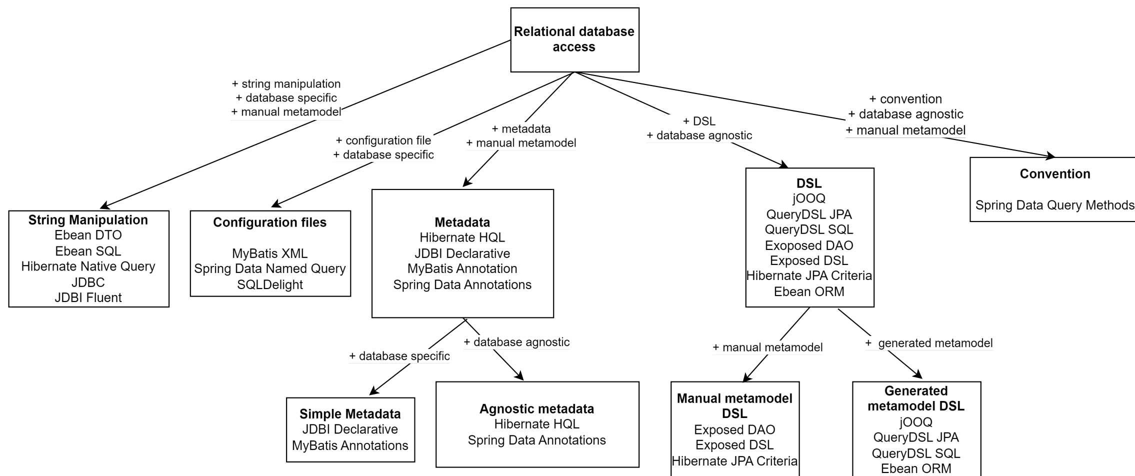

FIGURE 4.3: Overview of the various paradigms

exhibiting the three different types of properties. This overview is shown in Figure 4.3.

Chapter 5

Experiments

This chapter discusses the setup of the two experiments conducted in this thesis. Section 5.1 discusses the test business case used for both experiments. The first experiment consists of a performance measurement experiment, which is discussed in Section 5.2. Consequently, Section 5.2 contains the second experiment about code readability and complexity. The structure of both experiment sections are based upon the paper *"Reporting experiments in Software Engineering"* by Jedlitschka et al. [17].

5.1 Test business case

The goal of this experiment is to analyze the performance of the different implementation approaches as described in Section 4.4 in terms of execution time and power usage. In order to analyze and compare these performances of different implementation approaches, we need to create a sample business case which will be implemented for each of the implementations. The sample business case that will be used for this experiment originates from an entity relationship diagram (ERD) used at a project at *Info Support*. This ERD modelled the customer view and all entities related to the customer. Because of the big size of the original diagram, we have decided to use only a portion of it. This resulted in a smaller subset of the original diagram. Then, this subdiagram was converted to a UML class diagram as shown in Figure 5.1. Because the ERD did not contain any information about the properties of each entity, we have added a set of logically appropriate fields to each class.

The business case is centralized around the concept of an Entity: this can either be an individual, an organization or a department. Then, some other classes describe properties of these entities, like a name, address, digital address or document. Each of these data tables can then have a type again. Finally, among entities, different relations can be described. This business case can be challenging to properly implement in certain frameworks, because of the inheritance relationship between the entity class and the individual, organization, and department classes. This is a good, practical example of the impedance mismatch [29] problem as earlier described in Section 2.2.

This class diagram has then been converted to a SQL schema. This SQL schema has then been inserted into a PostgreSQL database, together with the correct primary key and foreign key configurations. Using a tool called *Java Faker*[1], we have generated an initial seeding of the database to run our data access operations on. We have chosen to use PostgreSQL as the RDBMS for this test business case because it is one of the systems

[1] https://github.com/DiUS/java-faker

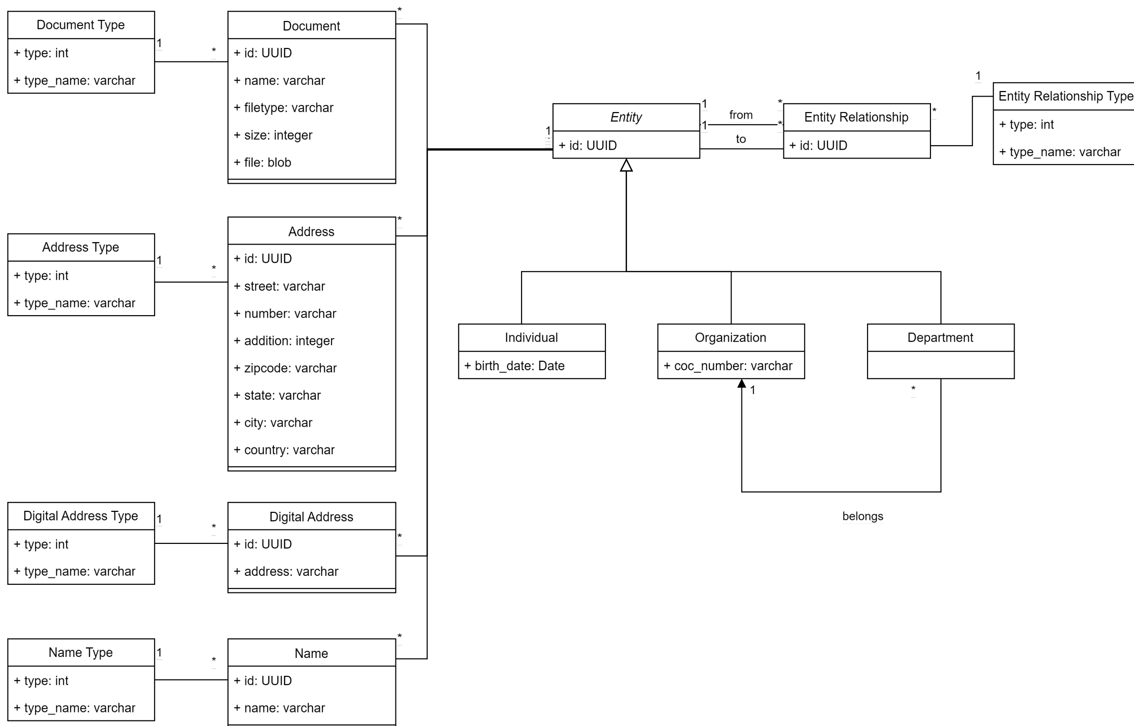

FIGURE 5.1: UML class diagram of the business case

supported by all of the frameworks under review, as shown in Table 3.2.

5.1.1 Data access operations

For every implementation, the same types of data access operations need to be specified, such that they can be compared both on performance and code readability. We have grouped the data access operations in two categories: simple and advanced operations.

Simple operations

We have defined five simple operations: two read operations and one each for insert, update, and delete.

1. **Individuals paginated read**. This operation requests a list of all individuals. Because this list can become large, it accepts two integer parameters: the page number and the page size. This structure is commonly used for larger data sets, as typically only a smaller subset can be displayed to the user at any given time.

2. **Individual with addresses**. This operation requires the UUID of an individual, and will then return the data of this individual, together with a list of all addresses linked to the individual.

3. **Inserting organizations with departments**. This operation gets a list of organizations as input, with for each organization a list of departments to be created. Also, for each department, a name is given, which is a separate record, as shown in the class diagram in Figure 5.1.

4. **Updating digital addresses of individual**. This operation has an map as input, with the key being the UUID of an individual, and as value the new digital address of

this individual. This data access operation should then update all digital addresses of this individual to the newly provided digital address.

5. **Deleting individuals from a country**. This operation deletes all individuals to which an address is linked of the specified country. In order to maintain database integrity, the SQL schema is set up in such a way that a cascade delete will occur properly.

Advanced operations

We have defined five advanced read operations.

1. **Individuals having a certain document type**. This data access operation returns a list of individuals, to which at least one document of the given type is linked. Also, the legal name and document size are returned.
2. **Addresses of individuals without digital address**. This operation retrieves a list of physical addresses of all individuals who do not have a digital address.
3. **(Digital) addresses of individuals with a document older than 10 years**. This operation requests a list of digital addresses individuals which have a document linked to them older than 10 years. In case there is no digital address known for the given individual, a formatted string of the postal address is returned instead.
4. **Individuals with invalid organization/department relationship**. This data access operation returns a list of individuals who have a relationship with both an organization and a department, where the department does not belong to the given organization. Different entities can be linked using the *EntityRelationship* table.
5. **Addresses linked to both an individual and an organization**. This operation retrieves a list of addresses linked to both an individual and an organization. An address can only be directly linked to a single entity, as shown in Figure 5.1. Therefore, this operation searches for address records with identical content (e.g., street, number, zipcode, etc.) that are associated with two different entities.

Implementation consistency

Each implementation approach has a different way of expressing the data access operations of the business case. Hence, it cannot always be immediately clear that all data access operations are implemented exactly the same, and are returning the same result sets. Hence, in order to make a fair and equal comparison among the different frameworks, we need to assure that the responses are consistent and equal for each of the operations.

We have solved this issue by taking the JDBC implementation as the baseline implementation. For every implementation approach, two interfaces are given which need to be implemented in that implementation: the `AdvancedDataAccessOperations` and `SimpleDataAccessOperations` interfaces. Also, we have created so called data transfer objects (DTOs) as the return type for each of the data access operations. This is a common pattern in Java, where the data requested is mapped to another object which will then be used to transfer to another host. Consequently, because every implementation utilizes the same interfaces, we can use this to verify the correctness of the implementation. We have selected the JDBC implementation to be the baseline implementation, which means that every implementation is compared with the JDBC implementation for correctness using unit tests.

5.2 Experiment I: Performance

This section discusses the experiment setup of the first experiment of this thesis: the performance measurement experiment.

5.2.1 Experiment design

For this experiment, we use the implementations of the different implementations for the test business case as described in Section 5.1.

5.2.2 Experimental Materials

The performance tests have been executed on a machine with an Intel Core i7-9750 @ 2.60Ghz with 6 cores, with 16GBs of RAM. This machine uses Ubuntu 22.04.4 LTS as the operating system, with Linux Kernel verison `6.5.0-44-generic`. We are running Ubuntu in recovery mode. This minimizes the number of background processes started, thereby reducing their impact on our performance measurements. The performance tests have been executed using OpenJDK 21.0.3, together with a PostgreSQL 16.3 database running inside a Docker container. Running this inside a Docker container does mean it incurs a performance overhead. [25]. However, this overhead is small and is present in all performance measurements, reducing the impact it will have on the performance measurements.

5.2.3 Hypotheses, parameters and variables

In this experiment, we measure the runtime in milliseconds, CPU package energy consumption in joules, and DRAM energy consumption in joules. We measure these variables for the different implementation approaches, as shown in Table 4.2. For each of these implementations, we execute the data access operations as described in Section 5.1.1; the five simple operations are measured separately because they represent different types of CRUD operations, while the advanced operations are measured together because they are all read operations. We expect that the implementations which are more more imperative and less abstract have a lower energy consumption and execution time than the implementations which are more declarative and more abstract, as visualized in Figure 4.2. We also expect the JDBC implementation to consume the least energy and has the lowest execution times. We expect that the DRAM energy consumption is less relevant, because we think that the efficiency of most implementations depend more on CPU bottlenecks rather than on RAM limitations.

5.2.4 Methodology

A shell script has been used in order to run the performance tests in an automated and consistent manner. First, a new Docker container is started and seeded with the initial data. This is needed because some frameworks require an active database connection in order to compile the code. Consequently, the main method of either the Simple or Advanced operations is executed.

First, the `DataAccessOperationsRunner` is started. This runner initializes the Docker container with the database and waits for it to be initialized. Then, the program will sleep for 5 seconds such that the processor usage will stabilize. Consequently, the operations as described in Section 5.1.1 will be executed. Then, this process is then repeated 13 times within the same JVM. The JVM uses just-in-time (JIT) compilation, which means

that the program can be dynamically optimized during execution. This means that a warm-up effect can occur, which means that the first few iterations can be slower than the succeeding ones. [23]. Hence, we discard the first three measurements, leaving us with 10 measurements per operation, per implementation to use for analysis.

Because jRAPL measures the energy consumption of the entire system, we need to account for any other processes which might be running on the system and hence also consume some energy. By measuring the energy consumed by the system while idle over a given time period, we can estimate how much energy consumption is unrelated to our performance test. Then, we can subtract this estimated idle load, such that we get an estimate of the actual amount of energy used by our test.

Then, we will group our measurements per data access operation, and take the average, median and standard deviation for each of the 10 measurements per implementation. Additionally, we will also determine the correlation between the runtime and CPU energy consumption and DRAM energy consumption.

5.2.5 Execution

During the implementation all the different data access operations, we encountered an issue with the Ebean framework. This framework does not support[2] the JPA inheritance strategies *JOINED* or *TABLE PER CLASS*, does not intend to support it in the near future. Since this is a key feature in the design of our database structure and we could not get an implementation working for Ebean, we have discarded Ebean from the list of frameworks under evaluation.

5.2.6 Analysis

Tables 5.1 up to and including 5.6 show the runtime and CPU & DRAM energy consumption per implementation. Finally, Table 5.7 shows the Spearman correlation coefficients between runtime & CPU energy consumption and runtime & DRAM energy consumption & runtime for each of the operations.

We have also calculated an overall ranking by summing the averages of the CPU energy usage and the runtime, and then sorting by the accumulated total of each of the operations per implementation. This gives us a better indication of how a framework performs across all data access operations. This is shown in Table 5.8 for the runtime and Table 5.9 for the CPU energy consumption.

Additionally, we can check whether there is a correlation between the CPU package energy consumption and the execution time, and the DRAM energy consumption and execution time. We will test this using the Spearman rank-order correlation coefficient.We will use the original data values for determining this correlation coefficient.

We can also determine whether there is a significant difference in performance of the various implementations based on their rankings. In order to test whether there is a significant difference, we apply the non-parametric Friedman test. This test is designed to compare three or more paired groups when the underlying distribution of the results is unknown. In our case, the different groups are the different implementations, which are tested against the different data access operations. If there is a significant difference, we can perform a Nemenyi-Friedman post-hoc analysis to see which implementations are significantly different from others.

Finally, we can test whether there is a significant difference in performance for each of the paradigms as distributed in Figure 4.3. For this, we utilize the Kruskal-Wallis H

[2] https://ebean.io/docs/mapping/

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	43259.4	42958.5	2841.0	778.50	765.32	71.83	33.38	33.19	1.78
Exposed DSL	52932.5	53168.0	2549.9	654.72	640.88	32.62	39.15	39.28	1.62
Hibernate HQL	2147.9	964.0	1869.6	23.11	23.46	3.60	1.34	0.62	1.14
Hibernate JPA Criteria	2445.0	1151.0	2089.9	30.40	27.18	9.35	1.60	1.07	1.24
Hibernate Native Query	2338.3	1158.0	1987.2	23.77	24.76	4.81	1.46	0.74	1.21
JDBC	1370.0	810.5	977.4	13.05	12.25	2.90	0.86	0.52	0.60
JDBI Declarative	41429.4	41581.5	1809.3	776.86	775.10	47.06	33.08	33.18	1.16
JDBI Fluent	39804.7	39901.0	1408.6	728.97	729.47	41.67	30.41	30.45	0.90
jOOQ	2984.9	2857.5	1909.2	18.51	18.46	3.38	1.88	1.80	1.16
MyBatis Annotations	1317.0	911.0	894.9	14.91	14.87	3.00	0.90	0.65	0.55
MyBatis XML	2394.5	1337.5	1803.8	31.80	31.88	5.21	1.66	0.99	1.13
QueryDSL JPA	1600.8	1378.0	714.9	36.48	35.74	6.79	1.03	0.89	0.44
QueryDSL SQL	2675.7	2356.5	1773.8	15.60	16.32	2.27	1.69	1.49	1.08
Spring Data Annotations	1280.0	842.0	998.5	15.98	16.42	2.50	0.82	0.55	0.61
Spring Data Named Queries	2427.1	2258.5	1404.8	15.29	15.69	2.13	1.52	1.41	0.86
Spring Data Query Methods	5896.8	5155.0	2856.9	84.22	80.77	12.85	3.88	3.43	1.76
SQLDelight	39632.4	39087.5	1675.2	757.31	776.47	51.64	30.39	30.02	1.04

Table 5.1: Performance measurements of the simple page read operation

test to see whether there are is a pair of paradigms which are significantly different from eachother. If this is the case, we carry out another post-hoc analysis to see which of the paradigms are significantly different. For this, we will use the Mann-Whitney U test with Bonferroni correction.

Simple page read operation

In Table 5.1, the results of the simple page read operations are shown. We can see that the JDBC implementation here has the lowest energy consumption, although other implementations also come close to this value. However, the MyBatis annotations implementation was slightly faster in terms of executon time. The Exposed DAO, Exposed DSL, JDBI Declarative, JDBI Fluent and SQLDelight implementations all were a lot slower and consumed more energy. Also, the standard deviation is in some cases a lot higher for the runtime measurements compared to the CPU energy usage.

The Spearman correlation between the runtime and CPU is 0.70118, and the Pearson correlation between the runtime and DRAM energy consumption is 0.99746, as shown in Table 5.7. This means that in all measurements, the runtime is strongly correlated with the CPU package energy consumption, and that the runtime is very strongly correlated with the DRAM energy consumption.

Simple nested read operation

In Table 5.2, the results of the simple nested read operations are shown. Here, we see that the Hibernate HQL and Hibernate JPA Criteria implementation are the fastest and consume the least CPU energy. Again, the JDBC implementation is not the fastest one as was expected initially, but was still reasonably fast compared to most other implementations. The Exposed DAO, Exposed DSL, JDBI Declarative, JDBI Fluent and SQLDelight implementations were the slowest, with their averages in runtime (25-30 seconds) and CPU energy consumption (3400-3600 joules) being in similar ranges.

The Spearman correlations as shown in 5.7 show a very strong and positive correlation of runtime & CPU and runtime & RAM (0.94959 and 0.99813, respectively).

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	294943.2	291940.5	8101.6	3591.70	3575.15	68.46	223.42	221.72	5.09
Exposed DSL	305593.6	276028.0	54096.7	3281.48	3264.17	145.20	229.46	210.62	34.41
Hibernate HQL	3026.9	2992.5	131.7	94.62	92.83	5.05	1.93	1.91	0.07
Hibernate JPA Criteria	4039.5	4116.0	314.5	120.89	123.74	9.22	2.83	2.78	0.34
Hibernate Native Query	12983.9	12504.0	1223.7	353.45	349.48	41.15	7.97	7.68	0.74
JDBC	11067.0	11052.0	891.9	283.94	283.49	18.39	6.74	6.73	0.54
JDBI Declarative	255775.8	257551.0	6987.1	3633.61	3617.98	95.80	203.57	204.69	4.43
JDBI Fluent	249569.5	251296.5	10499.2	3440.58	3438.18	169.59	191.63	192.81	6.77
jOOQ	14967.6	12889.5	3484.8	387.05	349.97	72.90	9.23	7.97	2.11
MyBatis Annotations	10577.9	10091.5	816.3	280.15	271.86	23.30	6.57	6.27	0.50
MyBatis XML	11532.2	11333.5	1123.0	311.79	300.23	26.62	7.09	6.97	0.68
QueryDSL JPA	15023.5	14688.0	987.8	412.58	401.29	27.03	9.62	9.42	0.60
QueryDSL SQL	14417.5	13023.0	3199.7	391.01	359.91	68.45	9.35	8.50	1.94
Spring Data Annotations	17437.0	16183.5	2667.6	473.91	473.31	19.56	10.89	10.12	1.63
Spring Data Named Queries	27814.9	27180.0	8785.4	475.29	458.54	79.63	17.19	16.84	5.33
Spring Data Query Methods	18401.2	19773.0	3796.1	478.98	457.28	75.61	11.39	12.22	2.31
SQLDelight	250856.4	250495.5	5514.4	3591.31	3615.52	115.32	193.01	192.66	3.46

TABLE 5.2: Performance measurements of the simple nested read operation

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	3278.3	3250.5	174.0	98.44	97.89	6.01	3.22	2.94	0.63
Exposed DSL	2530.3	2477.0	189.0	70.10	70.01	3.35	1.66	1.64	0.11
Hibernate HQL	305.6	302.0	24.8	10.06	9.94	2.11	0.24	0.23	0.07
Hibernate JPA Criteria	290.9	278.0	26.4	9.73	9.49	2.37	0.22	0.20	0.07
Hibernate Native Query	2279.0	1680.5	1154.1	49.68	50.28	3.59	1.47	1.10	0.70
JDBC	86120.7	85780.5	789.5	339.71	338.31	4.81	63.12	62.86	0.85
JDBI Declarative	89149.4	89192.0	425.2	360.30	360.75	3.43	66.92	66.94	0.26
JDBI Fluent	88505.2	88453.0	468.4	350.55	350.18	2.44	66.22	66.15	0.33
jOOQ	90122.9	90167.5	396.1	356.89	355.77	3.93	67.80	67.83	0.33
MyBatis Annotations	2373.7	2243.5	384.1	61.35	59.64	4.88	1.51	1.43	0.23
MyBatis XML	87212.7	87218.5	243.4	348.15	348.27	2.27	64.56	64.63	0.30
QueryDSL JPA	1595.6	1535.5	139.5	50.29	49.31	4.61	1.07	1.05	0.10
QueryDSL SQL	79798.1	79762.0	331.3	319.05	319.25	1.69	59.40	59.33	0.36
Spring Data Annotations	77372.4	77456.0	1130.0	326.03	326.72	4.84	53.13	53.40	1.27
Spring Data Named Queries	2807.5	2041.5	1643.2	50.46	49.09	4.21	1.81	1.33	1.02
Spring Data Query Methods	79210.3	79142.5	816.7	328.50	328.48	3.49	55.41	55.43	1.11
SQLDelight	612164.0	611552.5	4493.6	3196.65	3192.65	30.61	424.06	423.67	2.74

TABLE 5.3: Performance measurements of the simple insert operation

Simple insert operation

Table 5.3 shows that Hibernate HQL and Hibernate JPA Criteria are both very fast and consume the least energy. SQLDelight is the slowest and consumes the most energy by far: it is more than 6 times slower as the second slowest for this operation. We also noticed that the standard deviation of Hibernate Native Query and Spring Data Named queries are relatively high for the execution time. However, this is not the case when looking at the standard deviation of the CPU energy consumption.

The Spearman correlations as shown in 5.7 show a very strong and positive correlation of runtime & CPU and runtime & RAM (0.97631 and 0.99826, respectively).

Simple update operation

As shown in Table 5.4, the Exposed DAO and Spring Data Query Methods were the fastest and most energy efficient implementations. In this test, there were two implementations

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	26619.6	24841.0	7175.0	567.43	565.63	51.88	17.13	16.09	4.41
Exposed DSL	28125.1	27376.5	2678.4	729.86	721.25	58.15	17.88	17.49	1.64
Hibernate HQL	18014.2	17940.5	962.8	577.68	575.71	15.05	12.36	12.28	0.59
Hibernate JPA Criteria	23854.0	21958.0	4094.6	663.56	646.33	46.55	15.89	14.74	2.50
Hibernate Native Query	20540.0	20264.5	1719.7	608.46	604.13	49.14	13.20	13.01	1.07
JDBC	20453.5	18483.0	5135.0	528.89	513.46	56.16	12.56	11.36	3.14
JDBI Declarative	18607.2	18187.0	1789.9	592.51	590.74	43.93	11.82	11.56	1.10
JDBI Fluent	1661786.3	1664801.5	15503.8	28146.74	28167.16	391.81	1275.83	1278.39	11.20
jOOQ	19656.0	18799.0	2037.4	572.99	553.48	57.39	12.50	11.98	1.25
MyBatis Annotations	24499.7	23772.5	1976.9	703.06	691.90	47.61	15.39	14.94	1.21
MyBatis XML	19980.3	18190.0	5269.9	501.08	498.57	38.61	12.34	11.24	3.22
QueryDSL JPA	23939.0	23537.0	2339.6	634.40	625.42	47.27	16.06	15.93	1.44
QueryDSL SQL	24238.9	23048.0	6068.8	607.72	604.83	83.79	15.66	14.94	3.72
Spring Data Annotations	26750.1	26906.0	2288.7	695.60	690.81	37.32	18.22	18.29	1.43
Spring Data Named Queries	25387.2	23386.0	4006.3	642.21	633.29	40.63	17.00	15.76	2.47
Spring Data Query Methods	15550.2	14030.5	3623.3	390.56	391.45	12.02	9.92	9.00	2.22
SQLDelight	1653155.9	1654847.5	15231.6	28022.98	27968.52	501.17	1278.71	1281.46	9.57

Table 5.4: Performance measurements of the simple update operation.

which needed a lot more time and energy before finishing: JDBI Fluent and SQLDelight. These approaches are more than a hundred times slowed than the fastest implementation.

The Spearman correlations as shown in 5.7 are for runtime & CPU and runtime & RAM 0.78616 0.98957 respectively. This suggests a very strong and positive correlation.

As shown in Table 5.4, many of the implementations are in similar ranges in terms of performance.

Simple delete operation

For the delete operation, we see a big difference between the implementations in terms of execution time. However, this difference is a lot smaller when looking at the CPU energy usage. An explanation for this would be that upon deletion, the RDBMS might have been resolving some locking issues, which caused the thread to wait. This idle wait means that the execution time increases while the CPU energy consumption does not need to increase as much. However, it is difficult to test whether that is actually the case here.

The Spearman correlations as presented in Table 5.7 suggest a very strong and positive correlation between CPU energy usage & runtime, and between DRAM energy usage & runtime (0.88922 and 0.99878, respectively).

Advanced operations

In Table 5.6, the results of the advanced results are shown. We can see that on average, some of the frameworks happen to be faster than the JDBC implementation. We did not expect this result, because in our hypothesis we expected the JDBC implementation to be the fastest. However, six of the other implementations happened to be faster than the JDBC implementation. Also, ExposedDao and SpringDataQueryMethods are big outliers. For ExposedDao, the implementation is likely not efficient because of the *"data access object"* approach as shown in the name, because the joins to other tables happened very inefficiently here. For SpringDataQueryMethods, it was difficult to express the advanced operations in their method convention-approach, hence a lot of processing was required in Java, which in most other implementations happened in SQL or by the framework. JDBI also unexpectedly seemed to perform worse than other implementations here.

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	3677.9	3535.5	1020.2	27.89	27.86	1.35	2.50	2.42	0.64
Exposed DSL	3577.0	3564.0	315.1	17.95	17.72	1.34	2.48	2.45	0.19
Hibernate HQL	897.7	909.0	92.0	10.40	10.08	2.02	0.71	0.72	0.06
Hibernate JPA Criteria	879.1	890.5	46.6	10.76	10.18	2.14	0.70	0.70	0.03
Hibernate Native Query	760.6	768.5	29.7	3.31	3.27	0.15	0.54	0.55	0.02
JDBC	661.2	685.0	46.3	2.74	2.75	0.12	0.47	0.49	0.03
JDBI Declarative	3731.4	3717.0	285.8	18.55	18.52	0.28	2.59	2.59	0.18
JDBI Fluent	3648.0	3591.5	291.3	17.05	16.70	0.87	2.50	2.46	0.18
jOOQ	655.6	691.0	76.7	3.25	2.89	1.20	0.48	0.50	0.05
MyBatis Annotations	111.6	113.0	3.3	2.41	2.23	0.55	0.08	0.07	0.01
MyBatis XML	639.6	658.5	62.1	2.93	2.97	0.14	0.46	0.47	0.04
QueryDSL JPA	935.4	930.0	49.2	11.23	11.33	1.39	0.73	0.72	0.04
QueryDSL SQL	655.5	687.0	69.8	3.08	3.02	0.37	0.47	0.49	0.05
Spring Data Annotations	854.0	833.0	49.6	12.41	12.52	1.76	0.69	0.68	0.03
Spring Data Named Queries	953.2	944.5	55.1	10.82	10.80	1.50	0.75	0.74	0.03
Spring Data Query Methods	2471.6	2277.0	885.4	29.33	28.20	6.73	1.64	1.53	0.56
SQLDelight	3479.2	3553.0	308.0	18.28	18.51	0.64	2.40	2.44	0.19

TABLE 5.5: Performance measurements of the simple delete operation

	Runtime (ms)			CPU Package (J)			DRAM (J)		
Implementation	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**	**Avg**	**Med**	**Std**
Exposed DAO	448961.6	448026.0	2964.7	13031.71	13014.51	89.80	293.97	293.43	1.92
Exposed DSL	36942.0	36954.0	820.6	667.02	662.52	23.04	28.54	28.68	0.54
Hibernate HQL	5064.3	4949.5	281.6	134.19	135.97	5.65	3.23	3.18	0.16
Hibernate JPA Criteria	36975.7	36765.5	1304.1	803.53	784.05	52.89	23.04	23.01	0.78
Hibernate Native Query	5123.2	4853.0	816.0	133.63	128.26	20.44	3.25	3.07	0.49
JDBC	7880.1	7961.0	311.4	194.34	193.39	15.28	4.90	4.95	0.19
JDBI Declarative	28841.6	28800.0	605.1	618.02	626.33	18.99	23.81	23.73	0.52
JDBI Fluent	32251.7	32163.0	775.9	659.26	653.67	19.96	25.24	25.04	0.69
jOOQ	8391.7	8236.5	494.8	213.63	206.51	21.91	5.37	5.32	0.30
MyBatis Annotations	4260.9	4293.5	91.4	119.21	119.98	3.38	2.82	2.84	0.06
MyBatis XML	4573.2	4383.5	698.3	123.93	118.93	16.36	3.02	2.90	0.43
QueryDSL JPA	5328.9	5336.5	238.2	141.19	139.22	6.32	3.61	3.62	0.16
QueryDSL SQL	4604.6	4548.5	177.4	129.03	127.04	5.63	3.10	3.07	0.11
Spring Data Annotations	4434.7	4311.5	385.8	127.41	125.05	9.08	2.87	2.79	0.24
Spring Data Named Queries	4592.8	4479.5	388.4	110.83	103.14	14.51	2.97	2.89	0.25
Spring Data Query Methods	232401.8	233045.0	3888.3	6638.35	6662.47	95.35	152.63	153.11	2.40
SQLDelight	31999.7	32030.5	569.0	702.20	697.14	21.56	25.12	25.18	0.36

TABLE 5.6: Performance measurements of the advanced operations

Operation	Runtime, CPU	Runtime, DRAM
Page Read	0.70118	0.99746
Nested Read	0.94959	0.99813
Insert	0.97631	0.99826
Update	0.78616	0.98957
Delete	0.88922	0.99787
Advanced	0.95299	0.97952

TABLE 5.7: Spearman coefficients for each of the data access operations.

The Spearman correlations in Table 5.7 show that both runtime and CPU energy usage (0.95299), and runtime and DRAM energy usage (0.97952) are highly correlated.

Friedman-Nemenyi

Table 5.8 shows an overview of the averages of each of the implementations per data access operation. On this table, we can perform a Friedman test. This resulted in a p-value of 7.37e−7. This can be interpreted that there are certain implementations which consistently are ranked higher or lower than another implementation. Because of this significance in the Friedman test ($p < 0.05$), we can also perform a post-hoc analysis using Nemenyi's test. When performing this test, we could see that the overall fastest implementation, Hibernate HQL, was statistically different than the slowest 2 implementations, when using $\alpha = 0.05$. The slowest implementation, SQLDelight, was statistically different from Hibernate HQL, MyBatis Annotations, Hibernate Native Query and MyBatis XML.

We can apply the same analysis for Table 5.9, which shows an overview of the averages of each of the implementations per data access operation. When performing the Friedman test, we got a p-value of 2.09e−7, which means that there are certain implementations which consistently are ranked differently than another implementation. When performing the Nemenyi post-hoc test, we found that the second least energy consuming implementation, Hibernate Native Query, is statistically different than the 3 most energy consuming implementations, when using $\alpha = 0.05$.

Kruskal-Wallis

When grouping the totals of Table 5.8 into the paradigms as shown in Figure 4.3, we can test whether a given paradigm significantly different in terms of runtime than another paradigm using the Kruskal-Wallis test. This test resulted in a p-value of 0.730, which means that with $\alpha = 0.05$, we cannot say that there is a paradigm significantly different from another.

We can apply the same for the energy consumption of the results shown in Table 5.9. When performing the Kruskal-Wallis test, this resulted in a p-value of 0.732, which again means that there is no paradigm which significantly consumes a different amount of energy than another.

Ranking averages

We have ranked every implementation and then took the average per paradigm. We ranked the best implementation with 1 and the worst implementation with 17. In the case of the runtime measurements, the best measurement was the fastest. Trivially, in case of the CPU energy consumption measurements, the best implementation has the least energy

Implementation	Paradigm	Advanced	Page Read	Nested Read	Insert	Update	Delete	Total
Hibernate HQL	Agnostic metadata	5064.3	2147.9	3026.9	305.6	18014.2	897.7	29456.6
MyBatis Annotations	Simple metadata	4260.9	1317.0	10577.9	2373.7	24499.7	111.6	43140.8
Hibernate Native Query	String Manipulation	5123.2	2338.3	12983.9	2279.0	20540.0	760.6	44025.0
QueryDSL JPA	Generated metamodel DSL	5328.9	1600.8	15023.5	1595.6	23939	935.4	48423.2
Spring Data Named Queries	Configuration files	4592.8	2427.1	27814.9	2807.5	25387.2	953.2	63982.7
Hibernate JPA Criteria	Manual metamodel DSL	36975.7	2445.0	4039.5	290.9	23854	879.1	68484.2
MyBatis XML	Configuration files	4573.2	2394.5	11532.2	87212.7	19980.3	639.6	126332.5
QueryDSL SQL	Generated metamodel DSL	4604.6	2675.7	14417.5	79798.1	24238.9	655.5	126390.3
JDBC	String manipulation	7880.1	1370.0	11067.0	86120.7	20453.5	661.2	127552.5
Spring Data Annotations	Agnostic metadata	4434.7	1280.0	17437.0	77372.4	26750.1	854.0	128128.2
jOOQ	Generated metamodel DSL	8391.7	2984.9	14967.6	90122.9	19656.0	655.6	136778.7
Spring Data Query Methods	Convention	232401.8	5896.8	18401.2	79210.3	15550.2	2471.6	353931.9
Exposed DSL	Manual metamodel DSL	36942.0	52932.5	305593.6	2530.3	28125.1	3577.0	429700.5
JDDBI Declarative	Simple metadata	28841.6	41429.4	255775.8	89149.4	18607.2	3731.4	437534.8
Exposed DAO	Manual metamodel DSL	448961.6	43259.4	294943.2	3278.3	26619.6	3677.9	820740.0
JDBI Fluent	String manipulation	32251.7	39804.7	249569.5	88505.2	1661786.3	3648.0	2075565.4
SQLDelight	Configuration files	31999.7	39632.4	250856.4	612164.0	1653155.9	3479.2	2591287.6

TABLE 5.8: The runtime averages shown per operation, per implementation, sorted by the accumulated amount. All values are in milliseconds.

Implementation	Paradigm	Advanced	Page Read	Nested Read	Insert	Update	Delete	Total
Hibernate HQL	Agnostic metadata	134.19	23.11	94.62	10.06	577.68	10.40	850.06
Hibernate Native Query	String Manipulation	133.63	23.77	353.45	49.68	608.46	3.31	1172.29
MyBatis Annotations	Simple metadata	119.21	14.91	280.15	61.35	703.06	2.41	1181.10
QueryDSL JPA	Generated metamodel DSL	141.19	36.48	412.58	50.29	634.40	11.23	1286.16
Spring Data Named Queries	Configuration files	110.83	15.29	475.29	50.46	642.21	10.82	1304.91
MyBatis XML	Configuration files	123.93	31.80	311.79	348.15	501.08	2.93	1319.69
JDBC	String manipulation	194.34	13.05	283.94	339.71	528.89	2.74	1362.67
QueryDSL SQL	Generated metamodel DSL	129.03	15.60	391.01	319.05	607.72	3.08	1465.50
jOOQ	Generated metamodel DSL	213.63	18.51	387.05	356.89	572.99	3.25	1552.32
Hibernate JPA Criteria	Manual metamodel DSL	803.53	30.40	120.89	9.73	663.56	10.76	1638.86
Spring Data Annotations	Agnostic metadata	127.41	15.98	473.91	326.03	695.60	12.41	1651.34
Exposed DSL	Manual metamodel DSL	667.02	654.72	3281.48	70.10	729.86	17.95	5421.14
JDBI Declarative	Simple metadata	618.02	776.86	3633.61	360.30	592.51	18.55	5999.85
Spring Data Query Methods	Convention	6638.35	84.22	478.98	328.50	390.56	29.33	7949.94
Exposed DAO	Manual metamodel DSL	13031.71	778.50	3591.70	98.44	567.43	27.89	18095.66
JDBI Fluent	String manipulation	659.26	728.97	3440.58	350.55	28146.74	17.05	33343.14
SQLDelight	Configuration files	702.20	757.31	3591.31	3196.65	28022.98	18.28	36288.74

TABLE 5.9: The CPU energy consumption averages shown per operation, per implementation, sorted by the accumulated amount. All values are in joules.

Paradigm	Duration ranking average
Agnostic metadata	5.5
Generated metamodel DSL	7.7
Simple metadata	8.0
String Manipulation	9.3
Configuration files	9.7
Manual metamodel DSL	11.3
Convention	12.0

TABLE 5.10: Duration ranking average per paradigm, sorted from lowest to highest, where the lowest ranking is the best.

Paradigm	CPU energy consumption ranking
Agnostic metadata	6.0
Generated metamodel DSL	7.0
Simple metadata	8.0
String Manipulation	8.3
Configuration files	9.3
Manual metamodel DSL	12.3
Convention	14.0

TABLE 5.11: CPU energy consumption ranking average per paradigm, sorted from lowest to highest, where the lowest ranking is the best.

consumption. Table 5.10 and 5.11 show how the different data access paradigms rank in terms of runtime and CPU energy consumption, respectively.

5.3 Experiment II: Code Readability

This section discusses the experiment setup of the first experiment of the second experiment of this thesis: the code readability experiment. We again use the test business case for each of the implementation approaches as described in Section 5.1.1. The structure of this chapter is based upon the paper *"Reporting experiments in Software Engineering"* [17], just like Section 5.2.

5.3.1 Experiment design

For this experiment, we use the implementations for the different implementations of the test business case as described in Section 5.1.

5.3.2 Hypotheses, parameters and variables

We expect that frameworks which are more declarative in their notation to use up less lines of code and have a lower Halstead effort value (E) than the implementations which are more imperative, as shown in Figure 4.2. Because JDBC is the baseline and most barebone implementation, we expect it to score the worst. The independent variables of this experiment are the source code snippets per operation, per implementation. The dependent variables are the Halstead code complexity metrics η_1, η_2, N_1, and N_2 (Section 2.5.1). Additionally, we also determine the source lines of code (SLOC)[3], and we try to see

[3]Source lines of code refers to the lines of code in a program or code snippet, excluding lines that contain comments or only consist of whitespace characters.

whether there is a correlation between source lines of code and the Halstead Effort metric. We expect that there is a positive correlation between the Halstead Effort metrics and the source lines of code.

5.3.3 Methodology

As mentioned in Section 2.5, there are many different ways of utilizing code quality metrics in order to determine code readability. However, as mentioned, not all metrics are able to capture the notion of readabilty.

Other papers have shown examples of applying Halstead complexity metrics in practice. We have used the definition of an article by Wolle, where the author tried to analyze the correlation between Halstead metrics and lines of code [34]. In the article, all Java operators, separators and keywords belong to the operators set η, and all other tokens, like identifiers and literals, belong to the operands set N. We use the same definitions for calculating the Halstead metrics for Java, Kotlin and SQL based languages.

In Section 2.5.2, we raised the problem of determining code complexity metrics of programming languages embedded inside another. In the case of this research project, it is often a SQL-like language, embedded inside Java or Kotlin. In order to perform a good comparison between the different implementations, these embedded SQL-queries also contribute to the readability of the overall code. However, existing code readability metrics do not account for the embedding of one language within another.

Hence, in order to improve the utility of our code readability measurements, we have decided to apply the Halstead metrics in a manner that it can calculate these metrics for embedded languages too. That is why in this section, we propose a methodology to calculate Halstead complexity metrics for embedded programming languages.

If we take the code snippet of Figure 2.3 again from Section 2.5.2, we can distinguish two languages: Java and SQL. We call the language which embeds another language the **host language** (in this case, Java), and the language that is embedded the **embedded language** (in this case, SQL). For both the host and embedded language, we can determine the Halstead η and N sets using the known theory. Then, we can merge the sets of operators and operands of the two languages together in order to determine Halstead metrics for the two languages embedded inside of another.

It is important to note when two operands or operators are seen as equal, because the η sets only consist of the distinct operands and operators. We give the following definitions for operands and operators:

- An operator O_1 has the following properties:
  - *type*: the token type as labeled by a scanner/lexer
  - *lang*: the language of origin
- An operand O_2 has the same properties as O_1, but additionally has the following property:
  - *content*: the content of the token as scanned/lexed

When the properties of the operands or operators are equal, we regard them as equal for determining the η sets with distinct operands and operators.

For each of the implementations, we select all methods which belong to a certain data access operation. We created 6 groups of methods: one group for the advanced operations, and the other five for each of the simple operations as specified in Section 5.1.1. For each

of these methods, the Halstead N and η sets were calculated (including any embedded languages). Then, per group, these sets are merged together in order to determine the Halstead complexity metrics for each group separately. This approach means that we do not take JPA entity classes, data transfer object classes or other configuration classes into account; we focus specifically on the code which defines the data access operation.

As also mentioned by Wolle [34], a metric like lines of code also depends on the code style used by the programmer. That is why we have applied the Google Java Style Guide [4] for the Java frameworks, and the Kotlin Coding Conventions [5] for the Kotlin frameworks.

5.3.4 Execution

We started by taking the ANTLRv4 [22] grammar for Java [6] and Kotlin [7], such that we are able to properly parse the implementations of the data access operations. Then, for the Java and Kotlin grammars, we have implemented a program that can calculate the Halstead metrics of a given parse tree. After that, we used a PostgreSQL grammar [8] and HQL grammar [9] to also compute the Halstead metrics for a piece of SQL code. We have slightly modified the latter grammar such that the JPQL queries used in this research project also fit the HQL grammar. Finally, the last step is extending the aforementioned Java and Kotlin programs which calculate the Halstead metric to also evaluate string literals and text blocks inside the Java code, and see whether it is a parseable SQL string. If this is the case, the Halstead metrics of the nested SQL code snippet is computed and consequently, the sets of operators and operands are merged with the Halstead metric of the host language. Finally, we needed to specify which parts of the code of the implementation belongs to a certain data access operation. In a text file, we have specified the Java class names together with a list of methods which should be included when calculating the Halstead metrics.

5.3.5 Analysis

For each of the implementations, we determined the η and N values for the different operation types. Using these values, we can calculate the Volume and Difficulty metrics as explained in Section 2.5.1. Then, using these two metrics, we calculate the Effort metric, which is described as the effort required for the programmer to write or understand a piece of source code. In Table 5.12, the Halstead Effort metric can be found per implementation. Additionally, in Table 5.13, the source lines of code per implementation is shown. The highest and lowest values for each of the data access operations have been highlighted with green and red respectively. In Table 5.14 and 5.15, the sorted total Halstead Effort and SLOC per implementation can be found.

We can see that in 4 of 6 data access operations, the Spring Data Query Methods implementation had the lowest Halstead Effort value. In the other two cases, the Exposed DAO implementation was the fastest. For two data access operations, JDBC had the highest Halstead Effort value. In general, we can see that JDBC overall had a high Halstead Effort value compared to the other implementations.

[4] https://google.github.io/styleguide/javaguide.html
[5] https://kotlinlang.org/docs/coding-conventions.html#source-file-names
[6] https://github.com/antlr/grammars-v4/tree/master/java/java20
[7] https://github.com/antlr/grammars-v4/tree/master/kotlin
[8] https://github.com/antlr/grammars-v4/tree/master/sql/postgresql
[9] https://github.com/hibernate/hibernate-orm/tree/main/hibernate-core/src/main/antlr/org/hibernate/grammars/hql

When looking at the source lines of code in Table 5.13, we can see that there is some overlap in terms of best and worst scoring values when compared with Table 5.12.

We can calculate Spearman's coefficient between the Halstead Effort metric and the source lines of code using the values as shown in Tables 5.12 and 5.13. This results in a Spearman coefficient for every column in each of the tables, i.e., per data access operation. In Table 5.16, we can see that the page read, insert, and update operations are strongly, positively correlated. For the nested read, delete and advanced operations, there still is a positive correlation, but less strong as the aforementioned operations.

Friedman-Nemenyi

We can test whether there is a significant difference in effort and source lines of code of the various implementations based on their rankings. For this, we use the Friedman non-parametric test. For Table 5.12, this resulted in a p-value of 4.34e$-$5, which means that there is a significant difference between at least two of the implementations when using $\alpha = 0.05$. The Nemenyi post-hoc analysis showed that the implementation with the lowest overall effort value, Spring Data Query Methods, is significantly different than the implementations with the two highest effort values, Hibernate JPA Criteria and JDBC. Additionally, the analysis also showed that the implementation with the second lowest effort value, Spring Data Named Queries, was also significantly different from JDBC.

The same analysis can also be applied to the source lines of code in Table 5.13. The Friedman test returned a p-value of 1.07e$-$4. The Nemenyi post-hoc analysis showed that the JDBI Declarative implementation is significantly different than all three the Spring Data implementations. Additionally, JDBC is also significantly different than the Spring Data Query Methods.

Kruskal-Wallis

We can apply the Kruskal-Wallis test to see whether there is a paradigm is significantly different than another. This means that we group the measurements of the totals of Table 5.12. The Kruskal-Wallis test resulted in a p-value of 0.034, indicating that there is a statistically significant difference between the paradigms overall (at = 0.05). However, the Nemenyi post-hoc test did not reveal specific significant pairwise differences between paradigms, suggesting that the overall significance may originate from small differences across multiple paradigms.

The same analysis can be applied to the source lines of code from the totals of Table 5.13. Here, the Kruskal-Wallis test returned a p-value of 0.111, which again means that there is not a paradigm significantly different from another.

Ranking averages

For both Halstead effort and SLOC results, shown in in Tables 5.14 and 5.15 respectively, we have ranked each implementation. We ranked the best implementation with 1 and the worst implementation with 17. For Halstead effort and SLOC counts, less is better. Table 5.17 and 5.18 show how the different data access paradigms rank in terms of Halstead effort and SLOC, respectively.

Here, we see that the Convention paradigm overall scored best.

Implementation	Advanced	Page Read	Nested Read	Update	Insert	Delete
Exposed DAO	503066.4	6685.6	12765.3	2648.7	19063.1	1392.0
Exposed DSL	775132.6	8025.6	44552.2	4075.5	22144.5	2770.9
Hibernate HQL	460183.5	6055.7	12461.6	12830.3	12957.1	8208.3
Hibernate JPA Criteria	820202.8	7006.7	14359.1	9501.6	14691.9	12172.5
Hibernate Native Query	619156.0	7953.9	35988.8	4883.7	37015.3	7020.2
JDBC	758354.4	15742.4	40014.9	9471.3	78388.5	10429.8
JDBI Declarative	408608.8	9180.3	44364.5	6070.3	38401.7	9846.4
JDBI Fluent	642211.9	7310.6	43188.6	4256.7	41064.9	6192.0
jOOQ	446639.4	7724.0	27583.8	2955.9	24043.3	1453.7
MyBatis Annotations	351302.8	5378.5	40816.0	4163.7	41483.4	6413.5
MyBatis XML	329042.6	2957.5	35291.3	3405.3	30107.6	4656.9
QueryDSL JPA	505379.0	5444.2	23361.2	4885.7	13823.6	3708.7
QueryDSL SQL	491464.3	6715.7	25792.3	3972.5	22291.4	2980.6
Spring Data Annotations	387175.1	5209.0	12461.6	4638.7	10776.3	5496.3
Spring Data Named Queries	332402.7	3730.6	10388.0	3329.4	10834.1	3997.3
Spring Data Query Methods	251034.7	2796.9	6370.3	3527.4	10776.3	1856.9
SQLDelight	370250.6	10709.2	29597.1	2827.4	24584.6	10441.5

TABLE 5.12: Halstead Effort metric for each implementation per operation.

Implementation	Advanced	Page Read	Nested Read	Update	Insert	Delete
Exposed DAO	96	14	25	9	23	7
Exposed DSL	146	15	33	9	26	7
Hibernate HQL	43	8	24	12	21	10
Hibernate JPA Criteria	159	12	24	13	21	16
Hibernate Native Query	94	14	27	13	40	16
JDBC	110	20	35	16	54	12
JDBI Declarative	132	17	31	17	39	19
JDBI Fluent	109	16	27	12	40	14
jOOQ	129	12	44	10	31	9
MyBatis Annotations	41	8	28	10	34	8
MyBatis XML	88	9	29	10	35	11
QueryDSL JPA	205	14	31	13	21	16
QueryDSL SQL	200	14	32	11	30	14
Spring Data Annotations	41	8	22	12	19	10
Spring Data Named Queries	97	6	20	9	20	12
Spring Data Query Methods	105	6	17	11	19	8
SQLDelight	114	15	21	7	22	14

TABLE 5.13: The source lines of code (SLOC) used for each implementation per operation.

Implementation	Paradigm	Total Effort
Spring Data Query Methods	Convention	276362.6
Spring Data Named Queries	Configuration files	364682.1
MyBatis XML	Configuration files	405461.2
Spring Data Annotations	Agnostic metadata	425757.1
SQLDelight	Configuration files	448410.5
MyBatis Annotations	Simple metadata	449558.0
jOOQ	Generated metamodel DSL	510400.1
Hibernate HQL	Agnostic metadata	512696.5
JDBI Declarative	Simple metadata	516472.1
Exposed DAO	Manual metamodel DSL	545621.0
QueryDSL SQL	Generated metamodel DSL	553216.8
QueryDSL JPA	Generated metamodel DSL	556602.3
Hibernate Native Query	String manipulation	712018.0
JDBI Fluent	String manipulation	744224.7
Exposed DSL	Manual metamodel DSL	856701.2
Hibernate JPA Criteria	Manual metamodel DSL	877934.6
JDBC	String manipulation	912401.4

Table 5.14: Total Halstead Effort of all data access operations per implementation, sorted from smallest to largest.

Implementation	Paradigm	Total SLOC
Spring Data Annotations	Convention	112
Hibernate HQL	Agnostic metadata	118
MyBatis Annotations	Simple metadata	129
Spring Data Named Queries	Configuration files	164
Spring Data Query Methods	Convention	166
Exposed DAO	Manual metamodel DSL	174
MyBatis XML	Configuration files	182
SQLDelight	Configuration files	193
Hibernate Native Query	String manipulation	204
JDBI Fluent	String manipulation	218
jOOQ	Generated metamodel DSL	235
Exposed DSL	Manual metamodel DSL	236
Hibernate JPA Criteria	Manual metamodel DSL	245
JDBC	String manipulation	247
JDBI Declarative	Simple metadata	255
QueryDSL JPA	Generated metamodel DSL	300
QueryDSL SQL	Generated metamodel DSL	301

Table 5.15: Total source lines of code (SLOC) of all data access operations per implementation, sorted from smallest to largest.

Operation	Effort, SLOC
Page Read	0.859490343
Nested Read	0.651720394
Insert	0.944004469
Update	0.783944362
Delete	0.593843453
Advanced	0.475781818

Table 5.16: Spearman correlations of Halstead Effort and source lines of code (SLOC), per data access operation.

Paradigm	Halstead effort ranking average
Convention	1.0
Configuration files	3.3
Agnostic metadata	6.0
Simple metadata	7.5
Generated metamodel DSL	10.0
Manual metamodel DSL	13.7
String manipulation	14.7

Table 5.17: Halstead effort ranking average per paradigm, sorted from lowest to highest, where the lowest ranking is the best.

Paradigm	SLOC ranking average
Agnostic metadata	2.0
Convention	3.0
Configuration files	6.3
Simple metadata	9.0
Manual metamodel DSL	10.3
String manipulation	11.0
Generated metamodel DSL	14.7

Table 5.18: SLOC ranking average per paradigm, sorted from lowest to highest, where the lowest ranking is the best.

Chapter 6

Discussion

In this chapter, we will discuss the results from the experiments described in Sections 5.2 and 5.3. Also, different threats to the validity of these results are discussed.

6.1 Experiment I: Performance

The results of the first experiment revealed many interesting statistics. One of the things that is remarkable is that our hypothesis that JDBC would be the fastest implementation is incorrect. A reason for this could be that the frameworks which scored better in this performance experiment used some optimizations to improve on both duration and energy consumption. We know that some of the implementations make use of a so called “persistence context” (e.g., Hibernate). It is possible that some of the data access operations are initially only written to this persistence context, and occasionally these changes are committed all at once towards the database. This can have a positive influence in the performance measurements.

Also, some of the frameworks we initially expected to be faster due to their low level of abstraction, scored relatively bad. For example, both the JDBI implementations scored both quite low. We could not directly find a reason why these implementations were so much slower. A very rough guess would be that it is related to the performance of mapping of the result sets returned by the JDBC driver into the required data transfer objects.

Spring Data Query Methods and Exposed DAO scored relatively bad in terms of execution time and CPU energy consumption. This is because for these two implementations, it is difficult to entirely express the data access operation as proposed by the implementation approach. As a consequence of this, multiple smaller data access operations and additional processing was required in order to fully implement the required operation. This resulted into a negative performance impact for these implementations.

Additionally, the results of the delete operation are remarkable. When looking at Table 5.5, we can see that in the cases of Exposed DSL is almost as slow as Exposed DAO. But when looking at the energy consumption of the two implementations, we can see that they have about 10 joules of energy consumption difference. In the same table, other examples can be found as well. A reason for this difference could be that the duration is sometimes higher due to the process waiting for some locks of the database management system to resolve. During this waiting time, the CPU is not put under full load, hence resulting in a lower energy consumption, despite the longer execution time.

The Kruskal-Wallis test showed that there are no paradigms significantly different from each other in both duration and energy consumption. Given that each paradigm consists of only a small number of implementation, it is also unlikely that the test would show a

significant difference, because a small number of implementations means that there are fewer ranks to be considered in the Kruskal-Wallis test. This means that the ranks provide less information about potential differences within the paradigms.

6.2 Experiment II: Code Readability

The results of our second experiment reveal that the Spring Data Query Methods implementation is the winner in terms of the Halstead effort metric. For the source lines of code, it got ranked fifth. Overall, JDBC got ranked last in the Halstead effort metric, as we had expected in our hypothesis.

When comparing the rankings of the implementations by Halstead effort and source lines of code in Table 5.14 and 5.15, we can see that there are a few differences in the rankings of the two lists. When we look at Table 5.16, we can see that there is a positive Spearman correlation between Halstead effort and source lines of code, as anticipated in our hypothesis.

The Kruskal-Wallis test showed that there are no paradigms significantly different from each other in terms of Halstead effort and source liens of code. Again, as explained in previous section, we have performed this statistical test with very limited samples to rank, which means that the test is less likely to show a statistical difference.

6.3 Threats to validity

There are different types of threats which could impact the validity of the results as presented in Chapter 5. Below, we list some of the these threats to take into account when interpreting the results and conclusions presented in this thesis.

6.3.1 Internal validity

Measurement errors. As is visible in the results when looking at the standard deviations of the performance experiment, the measurements are sometimes not entirely consistent and can contain outliers. Although we have tried to minimize the amount of factors which can influence the performance during the performance tests, it is still possible that some background process in the host machine causes for unforeseen performance impact. This can be reduced by increasing the sample size, hence running tests even more.

Low statistical power. Because the groups (in our case, the paradigms) in the Kruskal-Wallis tests performed in Chapter 5 contained relatively few samples, it was less likely that the test showed a statistically significant difference between the groups.

6.3.2 External validity

Generalizability. The results in the performance experiment are directly related to the data access operations which are implemented for each implementation. These results do not guarantee that in other, real-life enterprise software systems, the performance of these frameworks will work similarly. Also, the way the tests are executed might not reflect a workload a system in production might get. Things like caching might increase the performance of these frameworks significantly.

The same thing applies for the code readability experiment. Because we have only evaluated a single business case with limited data access operations, it is difficult to say

whether the results reflect the readability of real-life enterprise software systems. Additionally, code readability also depends on how the software engineer implements a certain framework. For instance, an engineer might choose the most readable implementation identified in our research but implement it in a way that results in readability metrics that are even worse than those of the least readable implementation presented in this thesis.

6.3.3 Construct validity

Usage of metrics. In this study, we have utilized code complexity metrics in order to say something about the readability of a code snippet. Ideally, we would have performed a human study to see how software engineers rate different code snippets, instead of referring to a code metric. Due to time constraints, we have chosen to still use code metrics. Hence, the code complexity metrics as shown in this thesis should be regarded as a very rough estimate for determining code readability.

Optimal implementation bias. While implementing the test business case, we aimed to achieve the most optimal implementation possible in terms of performance and readability. However, it is possible that there exists a better solution in terms of performance or readability. Hence, this is an obvious threat to the reliability of the results.

Chapter 7

Conclusion

This chapter concludes with answering the research questions posed in Chapter 1. Below, we have listed the research questions again.

RQ1. What data access paradigms can be identified among the most popular data access frameworks for Java and Kotlin?

RQ2. What is the performance of the of an implementation of a data access paradigm?

(a) What is the performance of an implementation of a data access paradigm in terms of runtime?

(b) What is the performance of an implementation of a data access paradigm in terms of processor (CPU package) energy consumption?

(c) What is the performance of an implementation of a data access paradigm in terms of memory (DRAM) energy consumption?

RQ3. How readable is the source code of an implementation that is written using the a data access paradigm?

(a) What is the Halstead metric of the implementation of a data access paradigmk?

(b) How many source-lines-of-code (SLOC) does an implementation of a data access paradigm have?

7.1 Research Question 1

We have identified 11 different paradigms as shown in Figure 4.3. We have created a list of concepts which are able to distinguish the different implementation approaches into their respective paradigms. Below, a small summary is given of the data access paradigms which have been identified.

- **String Manipulation**. This paradigm is mainly focused on inserting parameters into SQL queries. It does not use any code generation based on an existing SQL schema nor is it database agnostic. It is the paradigm which leaves almost all decisions to the implementer and only assists in the formatting of SQL queries.
- **Configuration files**. The main focus of this paradigm is separating your data access operations from the rest of your business logic. This is done by putting these operations into separate files in your software project.

- **Metadata**. This paradigm is focused on specifying the data access operations as metadata in the programming language. In the case of Java, this additional metadata is provided using annotations above methods. The results of the data access operations are given as the return value of these methods. This paradigm also recognizes two subparadigms. The **simple metadata** subparadigm is database specific and the query specified in the annotation is executed on the database directly after inserting the required parameters. The **agnostic metadata** paradigm adds an additional abstraction layer by utilizing a query language like HQL or JPQL, making the database query agnostic for the database management sytem used.
- **DSL**. The name of this paradigm is an abbreviation of the well-known term *domain specific language*. It extends the host language by providing a type-safe API for specifying data access operations. A subparadigm of the DSL paradigm is the **manual metamodel DSL**, where the metamodel used for typechecks needs to be specified by the user, for example by specifying JPA entity classes. On the other hand, we have the **generated metamodel DSL** subparadigm, which generates these classes using the provided database schema.
- **Convention**. This data access paradigm uses a naming convention of which the data access operation to be executed is retrieved. This paradigm is database agnostic and requires the implementer to specify the metamodel manually.

7.2 Research Question 2

Section 5.2.6 has shown that there is no significant difference between data access paradigms in terms of execution time or energy consumption. We did show that the faster implementation, Hibernate HQL, was statistically different than the two slowest implementations, JDBI Fluent and SQLDelight. Also, we showed that the least power consuming implementation, Hibernate Native Query, consumed a statistically different amount than the three most power consuming implementations (Exposed DAO, JDBI Fluent and SQLDelight). Additionally, the results of the experiment in Section 5.2 can help a software architect in which data access paradigm to use when developing an enterprise software project.

(a) In terms of runtime, we can conclude from Table 5.10 that the "Agnostic metadata"-paradigm turned out to be the fastest paradigm in our measurements, followed by "Generated metamodel DSL", "Simple metadata", "String manipulation", "Configuration files", "Manual metamodel DSL", and finally, "Convention".

(b) In terms of CPU energy consumption, we can conclude from Table 5.11 that the least CPU energy consuming paradigm is "Agnostic metadata", followed by "Generated metamodel DSL", "Simple metadata", "String manipulation", "Configuration files", "Manual metamodel DSL", and finally, "Convention".

(c) We found that memory energy consumption is less relevant in the context of data access frameworks, because most data access operations demonstrated in the experiments were not memory intensive.

7.3 Research Question 3

Section 5.3.5 has shown that there is no significant difference between the data access paradigms in terms of the Halstead effort metric or the amount of source lines of code

used. We did show that the implementation with the lowest overall Halstead value, Spring Data Query Methods, is significantly different than the implementations with the two highest effort values, Hibernate JPA Criteria and JDBC. Also, the implementation with the second lowest effort value, Spring Data Named Queries, is significantly different from the implementation with the highest effort value, JDBC. Also, for source lines of code, we showed that the JDBC and JDBI Declarative implementations with the most lines of code are statistically different than the three Spring Data implementations. However, the results in 5.3 can give some indication to software architects how different implementations of the data access frameworks score relatively among each other.

(a) In terms of Halstead effort metric, we can conclude from Table 5.17 that the "Convention" paradigm has the lowest Halstead effort value, followed by "Configuration files", "Agnostic metadata", "Simple metadata", "Generated metamodel DSL", "Manual metamodel DSL", and finally, "String manipulation".

(b) In terms of source-lines-of-code (SLOC), we can conclude from Table 5.18 that the "Agnostic metadata" paradigm uses the least lines of code, followed by "Convention", "Configuration files", "Simple metadata", "Manual metamodel DSL", "String manipulation", and finally, "Generated metamodel DSL".

7.4 Future work

While this thesis aimed to address various aspects of data access frameworks, implementation approaches, and paradigms, there are still many other research directions to explore further. This section will briefly discuss other directions to be explored in the future.

7.4.1 Additional frameworks

As shown in Chapter 3, there are many different data access frameworks available. Because we had to set a limit to the scope of this research project, our selection included only 9 frameworks. However, as shown in Table 3.1, there are still some other frameworks remaining to be evaluated. Each of these frameworks can belong to any of the paradigms we have identified in this thesis, but could also form a new paradigm. Additionally, it would be interesting to see how these excluded frameworks perform in terms of execution time and energy consumption, and what their readability metrics are. Also, data access frameworks for other JVM languages like Scala or Groovy could be evaluated.

7.4.2 Non-relational database management systems

This thesis only considered data access frameworks with support for relational database systems. However, there are also database management systems which are non-relational. It would be interesting to see whether data access frameworks for these DBMSs also have similar paradigms as unveiled in this thesis, and whether there are maybe additional ones specifically for non-relational data access frameworks.

7.4.3 Additional business cases and operations

In this thesis, we have measured various performance and readability characteristics per implementation approach for only a single business case. In future research, additional business cases could be evaluated for these characteristics to see whether these results are in line with the results presented in this thesis. Also, real-life enterprise applications

using a given framework could be evaluated for the different characteristics to see how generalizable the results from this thesis are.

7.4.4 Discover additional concepts

In Chapter 4, we have introduced three concepts which we have used to classify the different types of paradigms. Maybe, additional and yet unknown concepts can be introduced which also have an influence on the way data access operations are specified.

7.4.5 Embedded language metrics

In Section 5.3.3, we have introduced a methodology for applying Halstead metrics to code snippets were one language is embedded into another. It would be interesting to see whether this can be applied to other readability metrics as well. Maybe, a general solution for the processing of embedded languages is desired.

7.4.6 Human evaluation

In this thesis, we have chosen to utilize code complexity metrics in order to reason about the readability of a code snippet. However, it would be very interesting to actually let software engineers annotate and/or rate the code snippets used in this research.

Bibliography

[1] Chitra Babu and G Gunasingh. Desh: Database evaluation system with hibernate orm framework. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2549–2556, 2016. doi:10.1109/ICACCI.2016.7732441.

[2] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95, 1995. doi:10.1109/WCRE.1995.514697.

[3] Alexandre Bonvoisin, Clément Quinton, and Romain Rouvoy. Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks. In Ipek Ozkaya and Fabio Palomba, editors, *31th IEEE International Conference on Software Analysis, Evolution and Reengineering - SANER 2024*, page 11, Rovaniemi, Finland, March 2024. IEEE. URL: https://inria.hal.science/hal-04401643.

[4] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery. URL: https://doi-org.ezproxy2.utwente.nl/10.1145/1390630.1390647, doi:10.1145/1390630.1390647.

[5] 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 of the 36th International Conference on Software Engineering*, ICSE 2014, page 1001–1012, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568259.

[6] 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, 2016. doi:10.1109/TSE.2016.2553039.

[7] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 165–176, 2016.

[8] Derek Colley, Clare Stanier, and Md Asaduzzaman. The impact of object-relational mapping frameworks on relational query performance. In *2018 International Conference on Computing, Electronics Communications Engineering (iCCECE)*, pages 47–52, 2018. doi:10.1109/iCCECOME.2018.8659222.

[9] Jonathan Dorn. A general software readability model. Department of Computer Science, University of Virginia, Charlottesville, Virginia, 2012. https://web.eecs.umich.edu/~weimerw/students/dorn-mcs-paper.pdf.

[10] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. Improving source code readability: theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 2–12. IEEE Press, 2019. doi:10.1109/ICPC.2019.00014.

[11] Siamak Farshidi, Slinger Jansen, and Mahdi Deldar. A decision model for programming language ecosystem selection: Seven industry case studies. *Information and Software Technology*, 139:106640, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0950584921001051, doi:10.1016/j.infsof.2021.106640.

[12] Martin Fowler. *Refactoring Improving The Design Of Existing Code*. 1999.

[13] Joe A. Garcia. Exploration of energy consumption using the intel running average power limit interface. In *2019 IEEE Space Computing Conference (SCC)*, pages 1–10, 2019. doi:10.1109/SpaceComp.2019.00005.

[14] Nikhil Govil. Applying halstead software science on different programming languages for analyzing software complexity. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, pages 939–943, 2020. doi:10.1109/ICOEI48184.2020.9142911.

[15] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., USA, 1977.

[16] International Organization for Standardization (ISO). NEN-ISO/IEC 25010:2023 systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models, 2023. NEN-ISO/IEC 25010:2023. URL: https://connect.nen.nl/Standard/Detail/3699042.

[17] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer London, London, 2008. doi:10.1007/978-1-84800-044-5_8.

[18] Johnson. Substring matching for clone detection and change tracking. In *Proceedings 1994 International Conference on Software Maintenance*, pages 120–126, 1994. doi:10.1109/ICSM.1994.336783.

[19] Weiyu Miao and Jeremy Siek. Compile-time reflection and metaprogramming for java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, page 27–37, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2543728.2543739.

[20] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359, 2020. doi:10.1109/ICSME46990.2020.00041.

[21] Oracle. Trail: The reflection api - (the java tutorials), n.d. Oralce Reflection. URL: https://docs.oracle.com/javase/tutorial/reflect/.

[22] Terence Parr. *The Definitive ANTLR 4 Reference, 2nd edition*. Pragmatic Bookshelf, 2013.

[23] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. *SIGPLAN Not.*, 49(10):345–360, oct 2014. doi:10.1145/2714064.2660235.

[24] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA, 2011. Association for Computing Machinery. URL: https://doi-org.ezproxy2.utwente.nl/10.1145/1985441.1985454, doi:10.1145/1985441.1985454.

[25] Eddie Antonio Santos, Carson McLean, Christopher Solinas, and Abram Hindle. How does docker affect energy consumption? evaluating workloads in and out of docker containers. *Journal of Systems and Software*, 146:14–25, 2018. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301456, doi:10.1016/j.jss.2018.07.077.

[26] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018. e1958 smr.1958. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1958, doi:10.1002/smr.1958.

[27] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016. doi:10.1109/ICPC.2016.7503707.

[28] Rares George Sfirlogea. A decision support model for using an object-relational mapping tool in the data management component of a software platform. 2015. URL: https://studenttheses.uu.nl/handle/20.500.12932/19405.

[29] Toby Teorey, Sam Lightstone, Tom Nadeau, and H.V. Jagadish. 8 - object-relational design. In Toby Teorey, Sam Lightstone, Tom Nadeau, and H.V. Jagadish, editors, *Database Modeling and Design (Fifth Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 139–160. Morgan Kaufmann, Boston, fifth edition edition, 2011. URL: https://www.sciencedirect.com/science/article/pii/B9780123820204000112, doi:10.1016/B978-0-12-382020-4.00011-2.

[30] Zoltán Tóth. Applying and evaluating halstead's complexity metrics and maintainability index for rpg. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Giuseppe Borruso, Carmelo M. Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, Elena Stankova, and Alfredo Cuzzocrea, editors, *Computational Science and Its Applications – ICCSA 2017*, pages 575–590, Cham, 2017. Springer International Publishing.

[31] Cătălin Tudose and Carmen Odubăşteanu. Object-relational mapping using jpa, hibernate and spring data jpa. In *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pages 424–431, 2021. doi:10.1109/CSCS52396.2021.00076.

[32] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621, 2009.

[33] Ervin Varga. *Unraveling Software Maintenance and Evolution*. Springer International Publishing, 2017. URL: http://dx.doi.org/10.1007/978-3-319-71303-8, doi:10.1007/978-3-319-71303-8.

[34] Björn Wolle. Statische analyse von java-anwendungen - einen sich lines-of-code-metrik und halstead-länge? *Wirtschaftsinformatik*, 45:29–49, 02 2003. doi:10.1007/BF03250881.