

Assignment report

by Salah Mohamed

Submission date: 02-May-2025 12:04PM (UTC+0100)

Submission ID: 257595033

File name: 58880_Salah_Mohamed_Assignment_report_664183_706873193.docx (269.55K)

Word count: 4863

Character count: 28917

Abstract

This report examines the basics of OOP, and how they are used today in software development, with a focus on web service development. Encapsulation, inheritance, polymorphism, and abstraction are defined first, along with the ways they improve software modularity, maintainability, and scalability. Following that, the report presents web services, emphasising RESTful architectures and demonstrating how OOP fits in well with these service models. The next section is a thorough analysis of ORM, particularly with Java and Hibernate, discussing how ORM facilitates object based programming techniques and streamlines database interaction.

The next section examines the Spring Boot framework, assessing the way it promotes modular and reusable code by incorporating typical design patterns like Dependency Injection and the Repository Pattern with OOP principles. Real world challenges like performance overhead, lazy loading, and data consistency problems are discussed along with best practices for creating scalable, effective web services with Spring Boot and ORM. The report ends with useful suggestions for enhancing ORM strategies in enterprise applications after examining several new trends, such as microservices, cloud native development, and GraphQL.

Table of Contents

Abstract	1
<i>Principles of Object Oriented Programming and Their Role in Software Design</i>	3
Introduction to Object Oriented Programming	3
Why Use OOP	3
Encapsulation	4
Inheritance	5
Polymorphism	6
Abstraction	7
<i>Overview of Web Services and RESTful Architectures in Modern Software Engineering</i>	8
Introduction to Web Services	8
Why do we use Web Services	8
RESTful Web Services	9
Key principles:	9
Why do we use REST Services	9
<i>Object Relational Mapping</i>	10
ORM Tools Hibernate	11
<i>Spring Boot and OOP</i>	12
How Spring Boot Integrates with OOP Concepts to Enhance Maintainability, Modularity, and Reusability	12
Evaluation of Design Patterns Used in Spring Boot and Their Alignment with OOP Principles....	13
Best Practices in Designing a Scalable and Efficient Web Service Using Spring Boot and ORM ...	14
<i>Challenges And Future Perspectives</i>	15
Common Challenges in OOP-Based Web Services Using ORM	15
Emerging Trends in Web Service Architectures and Their Implications on OOP and ORM.....	16
Recommendations for Improving ORM Strategies in Large-Scale Enterprise Applications	17
<i>Bibliography</i>	18

Principles of Object Oriented Programming and Their Role in Software Design

Introduction to Object Oriented Programming

Using objects to represent data and functions in software development is the main goal of OOP. Considering an object as a physical entity, like a car that has its own key attributes and methods it can perform. For instance, an object representing a car may have methods like accelerate and brake in addition to characteristics like colour and speed. Procedural programming, which primarily concentrates on functions and procedures instead of combining data and actions, is distinct from OOP. Making Application easier to comprehend, create, and maintain is the primary objective of OOP, particularly as they grow in size and complexity.

Why Use OOP

OOP was created to address issues with prior programming paradigms, such as procedural programming. It became challenging for developers to keep on top of all the variables, functions, and interactions as software projects got bigger. OOP makes the software considerably easier to use by combining related functions and data into one group. OOP is used for the following reasons:

- Handling complexity: OOP divides a large software into more manageable units.
- Promoting modularity: By allowing each object to be created and tested independently, teamwork is facilitated while efficiency is raised.
- Reusability: Developers save time and effort by using a class a blueprint for objects in different projects or applications after creating it.
- Scalability: OOP facilitates future program updates and the addition of new functionalities.
- Safety: OOP assists in protecting the application's integrity by regulating how data is accessed.

Key concepts of OOP:

Encapsulation, inheritance, polymorphism, and abstraction are the fundamental concepts upon which OOP is based. These are frequently referred to as the pillars of OOP.

Encapsulation

Encapsulation is the concept of limiting the use of a class to certain methods and keeping the details of how data is maintained or kept inside of the class private. This makes it simpler to comprehend ways to interact with the object and protects data from unexpected modifications [1]. for example, a number plate attribute of a vehicle class in a car park application. Since it serves as a unique identifier, the vehicle's number plate should not be altered once it has been made. By encapsulating the number plate, it is made private and only offer a way to view it and not alter it.

```
public class Vehicle 17 usages  🧑 salah mohamed
{
    private String numberPlate; 3 usages
```

This can stop anyone from accidentally altering this important attribute after the vehicle has entered the parking area by not defining a number plate set method. The class itself is in charge of managing this information, and the get method handles all logic that must read the license plate. Using encapsulation aids in avoiding errors. Additionally, encapsulation enables modifying the internal data later without impacting other code that utilises the vehicle class.

Inheritance

Inheritance enables building a new class from an existing one by taking over its methods and attributes. In this manner, a hierarchy is established, with more specialised features added to subclasses and general characteristics placed in superclass [2].

For example, in the car park scenario the superclass ID Readers has a common attributes and methods that are automatically extended by subclasses like ticket reader, barcode reader and each reader has unique characteristics but they both share certain capabilities.

```
public class IDReader 7 usages 4 inheritors 1 salah t
{
    private String serialNumber; 6 usages
    private int retryAttempts; 4 usages
    private boolean isInvalid; 3 usages
```

```
public class BarcodeReader extends IDReader 6 usages 1 salah t
{
    private boolean isTooFar; 3 usages
    private boolean isMisaligned; 3 usages
    private boolean backgroundInterference; 3 usages
```

```
public class TicketReader extends IDReader 6 usages 1 salah t
{
    private boolean isFolded; 3 usages
    private boolean isInsertedUpsideDown; 3 usages
    private boolean isUnreadableCode; 3 usages
    private boolean isJammed; 3 usages
    private boolean isInvalid; 3 usages
```

The use of inheritance may help in avoiding writing duplicate code and makes it simpler to manage shared logic. Only the superclass must be changed if there is a need to change how all readers execute the read method. Additionally, it makes the code more logically organised, making it easier for developers to comprehend how the code functions [2].

Polymorphism

Through the usage of a common interface or superclass, polymorphism enables referring to many objects that each implement distinct versions of particular behaviours. In real life, this implies that you may develop code that interacts with various kinds of things in a versatile and effective manner [3].

In the calculate fare method in fare class is implemented by all fare types, according to the Fare rule. Then different classes such as hourly weekly and monthly are created to represent different fares with each class using the calculate methods and overbidding to suit each payment type.

```
private static final double ratePerHour = 1.0; 1 usage
@Override 9 usages 1 salah mohamed
public double calculateFare(int hoursParked)
{
    return Math.max(1, hoursParked) * ratePerHour;
}
```

```
private static final double monthlyFare = 220.0;
@Override 9 usages 1 salah mohamed
public double calculateFare(int hoursParked)
{
    return monthlyFare;
}
```

This example of a payment calculations demonstrates how the use of polymorphism makes it simple to add new payment forms in the future and enables the parking system to handle various payment forms with ease. It illustrates the practical utility and flexibility of OOP polymorphism.

Abstraction

Abstraction is the process of designing a straightforward user interface that conceals intricate operational aspects. This enables users to use a class or object without having to fully understand how it functions internally [4].

The car park application has multiple display elements, including fare displays, full sign displays, and operating hours displays. Although the format and content will vary for each of these, they must all offer a means of displaying information to the user. An abstract class sign has a show method is used as a construction.

```
public abstract void show(); 6 usages 3 implementations 1 salah
```

The show method is now implemented differently by each display element based on the data it must present to the user.

```
@Override 6 usages 1 salah mohamed
public void show()
{
    if (!isActive)
    {
        System.out.println("Inactive, Sorry un
        return;
    }
    System.out.println("Fares:");
    System.out.println(" - £1 per hour");
    System.out.println(" - £60 per week");
}
```

```
@Override 6 usages 1 salah mohamed
public void show()
{
    if (!isActive)
    {
        System.out.println("Inactive, Sorry unable to show uot
        return;
    }
    System.out.println("Full sign: " + (lightOn ? "On" : "OFF");
}
```

Using abstraction is great as shown by the show method, it makes it easier to add new display types in the future such a discount display or a special hour display without altering the rest of your code.

An abstract show method enforces a contract each display must be able to display data, but each class is free to choose how to do so. This keeps the system adaptable and simple to expand [4].

Overview of Web Services and RESTful Architectures in Modern Software Engineering

Introduction to Web Services

Software systems hardly ever function in isolation in the connected world of today. They frequently interact and share resources throughout many platforms. One essential technology that makes this connectivity possible and simple to manage is web services which is an excellent way to facilitate communication between computers via a network, usually the Internet. Web services enable various programs connect with one another using standardised protocols and data formats [5].

Why do we use Web Services

- Enabling common web protocols such as HTTP to be used for communication between diverse systems.
- Supporting the expansion and transmission of software functionality over numerous servers and devices.
- Provide data or business logic so that numerous clients like web applications, mobile applications, and third parties can access it without having to rewrite complex code.
- Decoupling client implementation from server implementation to allow for independence system evolution.

Overall, web services are crucial for online shopping, cloud computing, applications for mobile devices, modern businesses, and IoT [5].

RESTful Web Services

Originally coined by Roy Fielding in his groundbreaking PhD dissertation, REST stands for Representational State Transfer. REST is an architectural approach used in network application architecture, it is not a protocol. Web services developed in accordance with the REST principles are known as RESTful web services. These services work with resources, which are listed as URLs, and communicate via the web, typically using HTTP [6].

Key principles:

- Every client, server request must include all the data required data for understanding and processing.
- Every resource can be accessed through a distinct URL.
- To manipulate resources, RESTful APIs employ the get, post, put, and delete standard HTTP techniques.
- Clients communicate with these representations rather than the core server data, and resources may have several formats, such as Json and xml.

Why do we use REST Services

- Improve performance and reduce latency and load on server, HTTP technologies like caching can be implemented to improve RESTful services.
- REST APIs makes it easier to make changes without breaking the clients as they use split the client and server.
- REST is widely used and drives nearly all cloud, mobile and web application becoming a standard for their many important benefits.
- REST makes it easier to scale throughout different systems and cloud environments.

RESTful web services are an important aspect in present software improvement as their usage is essential for serverless technologies, cloud based applications, and microservices architectures.

Also, they are considered as backbone of establishing an easy communication between systems developed by different businesses, promoting business agility and digital transformation. REST APIs makes automating testing, deployment process, and monitoring easier, which is essential for software best practises [6].

Object Relational Mapping

Object Relational Mapping is a software approach that enables software developers to effectively interact with a relational database in a simplified OOP manner. ORM simplifies the process by automatically transforming objects in code to rows in a database table, and back. Following this approach helps in lowering the need to manually construct repetitive SQL queries while focusing on the fundamental database operations like entering, updating, retrieving, and removing records [7].

In an ORM framework, class attributes correlate to table columns and object instances correlates to table rows. The primary objectives are to fill the gap between the OOP realm of programming and the relational realm of databases, which have mostly worked in distinct manner [7].

ORM is widely used as it brings many advantages such as making database common operations simpler, significantly improving software developers efficiency, and reducing boilerplate code while working with databases. ORM frameworks enables software developers to work more effectively within an object oriented paradigm as SQL queries is auto generated which minimises errors and inconsistencies that could potentially happen when moving between object models and relational schemas. Also, ORM improves maintainability by centralising mappings, making database schema changes simpler to handle without affecting the whole codebase. Furthermore, ORM helps protect against most vulnerabilities like SQL injection through the use of parameterised queries internally [8].

ORM is quite important in Java because the language is inherently OOP, whereas databases keep data in tables and rows. ORM frameworks in Java enable software developers to create entity classes, with each Java class representing a table in the database and each field representing a table column. Using frameworks such as Hibernate it automatically handles database sessions, transactions, object persistence, and the relationships.

Java specific ORM systems use @ annotation for declaring entity, table, id, and column to explicitly describe mappings. additionally, Java ORM tools help in the handling of complex object graphs, enabling associations such as one to many, many to one, and many to many to be handled simply within the code structure without having to manually handle any foreign keys [9].

ORM Tools Hibernate

Hibernate is widely known and used ORM frameworks in the Java realm. It enables software developers to interact effectively with databases using Java objects by abstracting away the complexities of JDBC. Handling the connection and the lifecycle of persistent objects, and offering an abstraction layer over SQL databases, Hibernate helps different applications be less reliant on a particular database provider [10].

Hibernate makes interacting with database simpler by enabling developers to concentrate on business logic rather than making and debugging SQL queries. It makes saving and deleting objects simple by using session save or session delete while Hibernate handles the basic SQL creation and connection to the database.

It adheres to OOP concepts like encapsulation and inheritance, enabling developers to express complicated domain logic concisely. It reduces boilerplate code and errors by using ready-made features such as handling transactions, caching, validation, and lazy loading.

Hibernate enables Java developers to build scalable and reliable applications with greater effectiveness than traditional JDBC, which is important for RESTful web services and microservices [10].

ORM concepts like Hibernate provide many significant advantages for when used for a web application. Firstly, by increasing software developer performance by lowering the need to constantly develop reparative SQL and enabling developers to interact with databases using typical object-oriented ways.

ORM when employed appropriately in large-scale application improves maintainability by centralising mappings and schema handling, which simplifies updates and version control. Also, it supports object-oriented principles like inheritance and polymorphism, as well as capabilities like caching, transaction management, and lazy loading, which, when properly configured, can optimise performance [11].

It comes with a few drawbacks, though a popular example is performance overhead, especially when interacting with large-scale data sets or complicated queries. Poor configuration can easily cause ineffective operation.

Also, database-specific features are managed less by developers, and ORM-generated queries are not always optimised. ORM may become constrictive in systems that need customised SQL. Although helpful, lazy loading can sometimes result in runtime problems if data is requested outside of active sessions [11].

Spring Boot and OOP

How Spring Boot Integrates with OOP Concepts to Enhance Maintainability, Modularity, and Reusability

Spring Boot is closely aligned with OOP concepts and provides an excellent structured and flexible process to developing contemporary, easy to maintain web applications. The architecture and concepts are based on modular, loosely linked elements that communicate via clearly defined interfaces, demonstrating key OOP concepts like encapsulation, inheritance, abstraction, and polymorphism.

it improves maintainability in multiple ways through using encapsulation and abstraction. A standard large scale project encapsulates important business logic in service classes and abstracts database activities behind repository interfaces. employing such structure separates the different layers of the application, when changes apply in one it rarely affects other layers. For example, a car service class that manages parking activities can use a car repository interface for getting data. Because the service only communicates with the interface, the implementation can be changed or replaced without affecting the business logic [12].

it encourages modularity via the using annotation and component scanning. classes annotated with Component, Service, or Rest Controller will be identified and registered as beans, enabling explicit separation. Each class serves a particular function and may be reused or tested individually [12].

Spring Boot strongly incorporates reusability through its use of interface driven development and inheritance. Developers are urged to make their code reusable in superclass and interfaces which enables common code to be abstracted and expanded if required. In RESTful API development, fundamental controller classes can handle common error replies and input validation, while individual controllers can be extended for unique use scenarios [12].

Furthermore, Spring Boot convention over configuration helps in lowering boilerplate whilst ensuring flexibility. Developers may override defaults when required, but lots of decisions regarding design adhere to best practices by default, improving consistency and lowering configuration errors. also, Spring Boot framework enables polymorphism, particularly in service classes that implements shared interfaces [13].

Overall, the framework demonstrates and strictly adheres to OOP concepts in a manner which promotes maintainability, modular structure, and implementing reuse. Its organised layering, component management, and interface driven architecture make it ideal for building large and scalable web applications.

Evaluation of Design Patterns Used in Spring Boot and Their Alignment with OOP Principles

the framework has multiple widely recognised design patterns which are adhering with OOP concepts, helping developers to create applications that promotes maintainability, modularity, and reusability. Two of the Spring Boot most popular patterns are Dependency Injection and the Repository Pattern. These patterns improve design by promoting loose coupling, separation, and the use of interface driven development which are all important OOP concepts.

Dependency Injection is a pattern in which a class gets its dependencies from an outside source instead of constructing them internally. The framework handles this automatically through the use of annotations such as `@Autowired`. Assume we have a library service class that must use a book repository. The framework makes a new repository into the service instead of simply building a new one which simplifies the service class a lot while adhering to the OOP concept of encapsulation, which keeps underlying functionality hidden and well organised. Also, it facilitates abstraction, as components frequently depend on interfaces instead of concrete classes. This means that updating or replacing elements of the system is simpler and done without affecting the rest of the code [14].

The Repository Pattern is a further vital part of the framework. Rather than creating code that will connect to the databases and execute SQL queries, you provide a repository interface with common method such as `find all`, `delete`, `save` etc., and Spring will generate the code for the implementation. This adheres to the OOP concept of abstraction, hiding complicated database logic behind interface [15].

Also, this pattern uses inheritance, as repository interfaces typically inherit a super interface, such as JPA Repository, which contains several common operations. It is not required to create code to get or save data since the system uses the logic described in the super interface.

The framework makes use of the Singleton Pattern for majority of its components. Generally, every part such as a service or repository is built once and distributed through the entire application which saves resources and guarantees that various elements of the application can collaborate on the same instance.

Best Practices in Designing a Scalable and Efficient Web Service Using Spring Boot and ORM

Using ORM and Spring Boot in designing scalable and effective web services requires following best practices that align with architectural principles and performance considerations. The goal is to maintain the system responsiveness, ease of maintenance, and scalability for future expansion the following points should be considered.

Implementing a layered design is widely recommended with explicit responsibilities for controller, service, and repository levels. Using this structure encourages loose coupling and is consistent with the Single Responsibility Principle which improves overall testability and modularity. Controllers, for example, manage request mappings, services maintain business logic, and repositories manage persistence.

Following interface driven development is additionally quite important developers can easily replace or extend components by writing against interfaces rather than implementations. Following this design adheres with the OOP concept of abstraction and simplifies unit testing.

ORM must be used to effectively manage entity relationships. Developers should consider using lazy loading for associations to minimise performance challenges such as the n+1 problem, in which recurring queries reduce performance. It is also recommended to employ DTOs to reduce the total amount of data exposed over APIs, which can notably improve both security and performance [7].

Transaction management is a good practice and must be used for consistency. The framework uses Transactional annotation to package operations that must work or not work together. This lowers the likelihood of incomplete updates and maintains the confidentiality of information in multi step procedures [8].

Performance tuning is another critical aspect that should not be ignored. Different tools such as Hibernate's second level caching, connection pooling, and batch operations for inserting and updating all help with improving the database performance. It is quite important to use these aspects especially in high load applications in which response time must be closely controlled [8].

Lastly, appropriate testing and monitoring is an essential part developers must create suitable unit and integration tests to cover all the elements, in addition to the framework Actuator, to monitor runtime behaviour. This enables early diagnosis of bottlenecks and improves over time reliability of the system [16].

Challenges And Future Perspectives

Common Challenges in OOP-Based Web Services Using ORM

Complex object relationships and automatically generated SQL can cause performance overhead in ORM frameworks like Hibernate. Inefficient joins and the N+1 query problem are two examples of problems where this is especially evident [8]. Despite the fact that these overheads exist, developers frequently give micro optimisation too much priority. Attempting to increase performance by 5–10% can take significant amount of time like days, weeks or months, and this rarely results in a significant impact.

Mostly raw ORM performance is rarely the bottleneck in modern systems, instead, it is i/o, network latency, or bad design. In these situations, developer time is worth more than slight speed increases. It is preferable to use tools like query logging or Actuator metrics to profile and optimise only where data clearly indicates problems [16].

Lazy loading is a further source of complications. This approach delays the loading of related objects until they've been explicitly accessed, potentially reducing the initial load time. However, if not properly handled, it can lead to the infamous n+1 selects problem, in which a distinct query is executed for each item in a collection. This can have a significant influence on the performance of databases, particularly when working with big data sets or complex relationships [16].

Another issue is data consistency. In OOP it is quite simple to manipulate objects without the need to instantly saving modifications made to the database which can result in discrepancies especially If transactions fail to be properly handled. For instance, if transactional boundaries are not properly used, partial changes can take place, resulting in a state that is inconsistent [17]. Furthermore, employing numerous data sources or concurrent requests raises the possibility of contradicting changes or outdated data.

ORM face other challenges when modelling complicated relationships like many to many or self referencing associations. The OO structure does not always map simply to tables with relationships, demanding extra work to ensure relational integrity and normalisation while preserving performance and readability [17].

Furthermore, ORM based solutions can be more challenging to debug and optimise. Since SQL queries are created at runtime, developers might have limited insight into how the system interact with the database unless they specifically enable query logging or using other tools.

Emerging Trends in Web Service Architectures and Their Implications on OOP and ORM

Typically, large web applications are divided into smaller independent services, every one of which is in charge of a distinct business function, according to the microservices architecture trend. Since every service encapsulates its own unique logic and information, this is consistent with the OOP concept of modularity. However, because each service usually has a unique interaction, microservices make it more difficult to use ORMs and discourage the system from using shared object relational mappings. Alternatively, developers might have to create Apis that translate among internal entities and shared representations or duplicate data models [18]. Although data consistency and cross service relationships need to be addressed explicitly, frequently with the use of asynchronous messaging or eventual consistency models, ORM is still quite valuable within each service.

The transition from RESTful service structures to more adaptable client driven data queries is reflected in GraphQL Apis. With GraphQL, clients are able to request precisely which data they require, in contrast with REST, in which endpoints return fixed structures of data. Developers must now create adaptable, embedded DTOs rather than directly exposing entities, which puts more pressure on the object model. GraphQL can map ORM entities selectively or completely avoid them, which increases complexity but decreases tight coupling. This change adapts the use of OOP to new data access patterns by promoting a more compositional design rather than rigid class inheritance [19].

Scalability, automation, and resilience are key components of cloud-native development. Applications are anticipated to scale easily based on requirements and are built to operate in containerised environments such as Kubernetes. This has an impact on ORM by promoting stateless, transactional operations and discouraged lengthy stateful sessions. In distributed environments, ORM need to be properly configured to prevent caching or session-related problems.

Recommendations for Improving ORM Strategies in Large-Scale Enterprise Applications

It is necessary for developers to embrace different strategies that regulate adaptability, efficiency, and maintainability in order to make ORM like Hibernate efficient for large scale projects.

applying DTOs rather than directly exposing entities is one important recommendation. Because of lazy associations, returning complete entity objects may result in tight coupling and unexpected loading of data. DTOs can significantly help lower API design risk and provide greater control over shaping data. It's also quite important to use explicit queries. Although automatically generated queries are supported by the framework and hibernate, effectively structured JPQL or native SQL may be used for essential tasks to improve readability as well as efficiency [7].

Careful management of fetching strategies is necessary as the N+1 issue may arise from the default lazy loading, particularly when it comes to collections. To optimise queries batch sizes or join fetching should be used effectively. Inefficiencies can be identified with the use of tools such as Hibernate statistics and query logging.

Finally, transaction boundaries need to be brief and precise. At the service layer, Transactional need to be used instead of retaining transactions across several operations or outside calls, which raises contention and runs a chance of inconsistency.

Bibliography

- [1] J. Micallef, "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages," Jan. 1987, doi: <https://doi.org/10.7916/d8tt4zzd>.
- [2] E. Tempero, J. Noble, and H. Melton, "How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software," *ECOOP 2008 – Object-Oriented Programming*, pp. 667–691, doi: https://doi.org/10.1007/978-3-540-70592-5_28.
- [3] N. Milojkovic, A. Caracciolo, M. F. Lungu, O. Nierstrasz, D. Rothlisberger, and R. Robbes, "Polymorphism in the Spotlight: Studying Its Prevalence in Java and Smalltalk," *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, doi: <https://doi.org/10.1109/icpc.2015.29>.
- [4] M. Lavand and R. Pawar, "A Study On Abstraction in Object Oriented Programming with Java and Its Implementation in Developing Programs," *International Journal on Science and Technology (IJSAT)* IJSAT25023438, vol. 16, no. 2, 2025, Available: <https://www.ijSAT.org/papers/2025/2/3438.pdf>
- [5] P. Muschamp, "An introduction to Web Services," *BT Technology Journal*, vol. 22, no. 1, pp. 9–18, 2004, doi: <https://doi.org/10.1023/b:bttj.0000015537.71918.c2>.
- [6] S. Tilkov, "A Brief Introduction to REST." Available: <https://www.espinosa-oviedo.com/web-programming/files/readings/A-Brief-Introduction-to-REST.pdf>
- [7] C. Ireland and D. Bowers, "Understanding Object-Relational Mapping: A Framework Based Approach," 2009.
- [8] V. Sivakumar, "Object Relational Mapping Framework Performance Impact," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 7, pp. 2516–2519, Apr. 2021, doi: <https://doi.org/10.17762/turcomat.v12i7.3580>.
- [9] O. Edward Erhieyovwe, N. Oluwole, and P. Oghenekaro, "An Object Relational Mapping Technique for Java Framework."
- [10] C. Babu and G. Gunasingh, "DESH: Database evaluation system with hibernate ORM framework," *IEEE Xplore*, Sep. 01, 2016. https://ieeexplore.ieee.org/abstract/document/7732441?casa_token=1THoDuhJ9xUAAAAA:focWqiv2omr6TlAi1BcoaKP0MdnSATWb9FrbrHAFpkBEU6nbDxBtE6MYpgDNIDHjr4eJbUYy44.
- [11] Kisman and S. M. Isa, "Hibernate ORM query simplification using hibernate criteria extension (HCE)," *IEEE Xplore*, Sep. 01, 2016. <https://ieeexplore.ieee.org/document/7725656>
- [12] M. Jeleń and M. Dzieńkowski, "The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus," *Journal of Computer Sciences Institute*, vol. 21, pp. 287–294, Dec. 2021, doi: <https://doi.org/10.35784/jcsi.2724>.

- [13] M. Mythily, A. Samson Arun Raj, and I. Thanakumar Joseph, "An Analysis of the Significance of Spring Boot in The Market," *IEEE Xplore*, Jul. 01, 2022. <https://ieeexplore.ieee.org/document/9850910>.
- [14] C.-W. Sun and C.-F. Liao, "Examining the Influence of Dependency Injection on Software Maintainability," *2023 IEEE/ACIS 8th International Conference on Big Data, Cloud Computing, and Data Science (BCD)*, pp. 250–255, Dec. 2023, doi: <https://doi.org/10.1109/bcd57833.2023.10466290>.
- [15] X. Luan, "IMPLEMENTATION AND ANALYSIS OF SOFTWARE DEVELOPMENT IN SPRING BOOT," 2021. Available: <https://scholarworks.calstate.edu/downloads/zg64ts132>
- [16] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *Information and Software Technology*, vol. 82, pp. 1–18, Feb. 2017, doi: <https://doi.org/10.1016/j.infsof.2016.09.009>.
- [17] M. Felipe, E. Guerra, R. Bonifacio, and G. Pinto, "Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report," *Biblioteca Digital da Memória Científica do INPE (National Institute for Space Research)*, Sep. 2019, doi: <https://doi.org/10.1109/esem.2019.8870163>.
- [18] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, Jan. 2018, doi: <https://doi.org/10.1145/3183628.3183631>.
- [19] Y. Marchuk, I. Dyyak, and I. Makar, "Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches," Sep. 2023, doi: <https://doi.org/10.1109/elit61488.2023.10310748>.

Assignment report

GRADEMARK REPORT

FINAL GRADE

70/100

GENERAL COMMENTS

A fairly good report. There are a few minor spelling, punctuation and grammar errors, however - be sure to check your work carefully.

A good use of sources.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

