

Critical analysis of the relationship between OOP and Web

Services_RyanEvans_2420520

by Ryan Evans

Submission date: 02-May-2025 02:25PM (UTC+0100)

Submission ID: 257628680

File name:

58749_Ryan_Evans_Critical_analysis_of_the_relationship_between_OOP_and_Web_Services_RyanEvans_2420520_664183_960578400.docx
(331.87K)

Word count: 6889

Character count: 42710

Critical analysis of the relationship between OOP and Web Services

Contents

Introduction	4
Conceptual Foundations	5
Object-Oriented Programming (OOP) Principles.....	5
Encapsulation.....	5
Inheritance.....	5
Polymorphism.....	6
Abstraction.....	6
Web Services and RESTful Architectures.....	7
Overview of Web Services.....	7
Types of Web Services	7
SOAP (Simple Object Access Protocol) Web Services:.....	7
REST (Representational State Transfer) Web Services:	8
Characteristics of Web Services	8
Interoperability	8
Standardised protocols.....	8
Platform and language independence	8
Loose coupling	8
Self-describing services	8
Scalability and reusability	9
Statelessness	9
Object-Relational Mapping (ORM) in Web Services.....	10
Object-Relational Mapping (ORM).....	10
Key purposes of ORM:	10
ORM Tools and Hibernate	10
Key features of Hibernate:.....	11
Why use Hibernate?	11
How ORM Tools Simplify Database Interactions within OOP.....	11
Seamless object-table mapping:	11
Automated data operations:	11
Relationship management:.....	12
Database abstraction:	12
Cleaner, maintainable code:.....	12

Advantages and Limitations of ORM	12
Reduced Boilerplate Code	12
Improved Maintainability.....	12
Abstraction from SQL	12
Limitations of Using ORM Tools.....	13
Performance Issues.....	13
Complexity with Large Datasets	13
Limitations in Query Optimisation.....	13
Spring Boot and OOP	13
Integration of OOP Concepts in Spring Boot.....	13
Design Patterns in Spring Boot.....	15
Dependency Injection	15
Repository Pattern.....	15
Enhancing Design Quality	16
Improving Efficiency	16
Best Practices	17
Apply Domain-Driven Design (DDD)	17
Implement Proper Transaction Management.....	17
Design a Modular Architecture	18
Optimise ORM Usage	18
Challenges and Future Perspectives	19
Common Implementation Challenges	19
Performance Overhead	19
Lazy Loading Issues.....	19
Data Consistency	19
Emerging Trends	20
Microservices.....	20
Cloud-Native Development	20
GraphQL APIs.....	20
How These Trends Might Evolve the Use of ORM.....	21
Smaller Scope, Focused Use:	21
Increased Hybrid Approaches:	21
Shift Toward API-Centric Thinking:.....	21
Recommendations	21
Performance Tuning	21
Strategic Caching	21
Query Optimisation.....	22
Transaction and Connection Management	22

Architectural Considerations 22

Conclusion 23

References..... 24

Introduction

Object-Oriented Programming (OOP) has long been a cornerstone of software development, offering a clear way to organise and manage complex systems by breaking them down into interacting objects. With principles like encapsulation, inheritance, polymorphism, and abstraction, OOP helps developers build software that is easier to maintain, extend, and understand.

In today's development landscape, these principles are put to work in powerful frameworks such as Spring Boot, which promotes modular design and simplifies the creation of scalable applications. Alongside this, web services particularly RESTful architectures allow systems built in different languages or platforms to communicate smoothly, while Object-Relational Mapping (ORM) tools like Hibernate make working with databases more straightforward by bridging the gap between object-oriented code and relational data.

This paper looks at how these concepts come together in modern software development, exploring both their advantages and the challenges they bring, and considering how emerging trends are shaping the way developers apply them.

Conceptual Foundations

Object-Oriented Programming (OOP) Principles

Encapsulation

Encapsulation constitutes a fundamental principle in OOP, which restricts access to an object's data and methods[1]. The primary objective of encapsulation is to protect the object's state from unintended modification, permitting alterations solely through defined methods, such as getters and setters[2].

Key aspects of encapsulation:

Data hiding: Variables within classes are customarily declared as private, thereby ensuring isolation from extrinsic factors[3].

Controlled Access: Public methods facilitate regulated access to the internal state, thereby preserving data integrity[2].

Simplified Maintenance: Through the utilisation of interfaces and abstract classes, encapsulation effectively minimises the complexity involved in maintaining and updating code[4].

Benefits of Encapsulation

The advantages of encapsulation include enhanced security and integrity of an object's state by preventing unintentional access or modification[3]. Furthermore, it simplifies interactions between components by abstracting internal data and behaviours behind well-defined interfaces. Lastly, it provides flexibility and maintainability, allowing internal changes without affecting the external code that relies on the class[4].

Inheritance

Inheritance is a fundamental concept in OOP, where one class, known as a subclass or child class, inherits the properties and behaviours of another class, called a superclass or parent class. This approach enables code reuse, supports hierarchical classification, and facilitates polymorphic behaviour within the system[5].

Key points of Inheritance

Reusability: Functionality that is commonly defined in a superclass can be reused across multiple subclasses[6].

Extensibility: Subclasses have the capability to introduce new features or to override existing behaviours of the superclass[7].

Hierarchy: Establishes an "is-a" relationship. For instance, a Car is a Vehicle.

Benefits of Inheritance

The advantages of inheritance encompass the reusability of code through the use of subclasses that extend existing implementations, thereby reducing redundancy and minimising duplication[6]. Furthermore, it supports scalability by enabling subclasses to incorporate additional functionalities without adversely affecting the superclass[7]. Lastly, it facilitates polymorphism by allowing subclasses to be treated as instances of their superclass, promoting generalisation and enhancing flexibility within the codebase.

Polymorphism

Polymorphism is a fundamental principle of OOP, which allows variance in behaviour from a single source. It allows different responses for the same method from various classes, each with its own functionality. This increases flexibility and allows for dynamic method binding[8].

Key points of Polymorphism

Dynamic Behaviour: Depending on the specific class necessitated during runtime, varying outcomes may be attained through the identical method invocation[9].

Method Overriding: Methods inherited from subclasses may be adjusted to deliver the specific functionality required at runtime[10].

Method Overloading: Within a class, numerous methods sharing a name can be utilised with varying parameters, allowing for polymorphism at compilation[11].

Generalisation: The functionality required within a subclass can vary from that of its superclass or interface[8].

Benefits of Polymorphism

The advantages of polymorphism include improved flexibility by allowing a single interface to work with different types of objects, thereby enhancing code generalisation and promoting reusable methods through a shared reference[8]. Furthermore, it supports maintainability and scalability by enabling the introduction of new object types with minimal impact on existing code, fostering an adaptable and robust software design[11].

Abstraction

Abstraction is a fundamental principle of OOP, where the implementation details are obscured from the user, with only the required functionality presented. This removes the focus from how an object works and promotes what the object does. Allowing for modularity within the system.

Key points of Abstraction

Data Hiding: The implementation functionality is obscured, ensuring the user only interacts with the essential components[3].

Interface Exposure: The necessary methods required by the user are exposed using public interfaces and abstract classes[2].

Simplification: The simplification of the system for users and developers is increased by abstracting the implementation functionality, Improving usability and maintainability[4].

Security and Control: By restricting the implementation functionality, the risk of unintended modifications is reduced, improving the consistency of the system[8].

Benefits of Abstraction

Abstraction offers significant advantages to the usability of the system by presenting the user with exclusively the necessary operations[2]. Furthermore, it enhances the security and integrity of the system by obscuring and regulating access to the internal logic[3]. Finally, maintainability and scalability are improved by ensuring a clear distinction between the internal implementation and the external code that depends on it, thereby permitting modifications or maintenance to be executed with minimal impact.

Web Services and RESTful Architectures

Overview of Web Services

Web services constitute a standard software component that facilitates communication and data exchange among diverse applications over a network[12]. These services operate independently of the underlying technologies, programming languages, or platforms. This machine-to-machine interaction adheres to a client-server architecture[13] and employs protocols such as HTML, XML, SOAP, and REST[14]. Web services typically utilise Application Programming Interfaces (APIs) to expose functionality to external clients through requests and responses.

Types of Web Services

SOAP (Simple Object Access Protocol) Web Services:

- Use XML-based messaging.
 - Operate over HTTP, SMTP, etc.
 - Are strict and highly standardised.
-

REST (Representational State Transfer) Web Services:

- Use HTTP methods (GET, POST, PUT, DELETE).
- Communicate via plain text, XML, or JSON (commonly JSON).
- They are lightweight, stateless, and easier to integrate with.

Characteristics of Web Services

Interoperability

Interoperability is one of the biggest advantages of web services. They let applications written in completely different programming languages and running on different platforms talk to each other smoothly[15]. For example, you can have a Java application on a Linux server exchanging data with a .NET app running on Windows. This is highly beneficial in organisations that have a mix of old and new systems.

Standardised protocols

Standardised protocols like HTTP, XML, and JSON are what make this possible. These widely accepted standards define how data is sent and received between systems[14]. By sticking to these, web services ensure that data is handled consistently and reliably, no matter what technologies are used on either side. This saves developers from needing to build custom or proprietary solutions just to get systems to communicate.

Platform and language independence

Platform and language independence means that the client and the server don't need to be built using the same tools[16]. A Python backend can serve data to a JavaScript front end, or a PHP client can call a .net service, if they both follow the same web protocols. This gives development teams more freedom to use the best tools required without concerns of compatibility[16].

Loose coupling

Loose coupling is a key design strength. It means the client and server are connected only by a shared interface, not by their inner workings[17]. While the contract stays the same, either side can be updated or replaced without breaking things. This makes the system easier to maintain and adapt over time.

Self-describing services

Self-describing services come with built-in documentation that tells you what they do and how to use them[18]. SOAP services, for example, provide a WSDL file that explains what methods are available and what kind of data they expect[19]. REST services often use something like an OpenAPI formerly Swagger definition for the same purpose[20]. This means developers don't need to dig into the code or guess how to interact with a service.

Scalability and reusability

Scalability and reusability are major practical benefits. A well-designed web service can be reused across multiple applications, saving time and reducing duplicate work[21]. And because each service is its own independent piece, it can be scaled as needed to handle increased demand, without affecting the rest of the system.

Statelessness

Statelessness, especially in RESTful services, means the server doesn't remember anything about past requests[22]. Each request carries all the information the server needs to process it. This simplifies the server's job and makes it easier to balance the load across multiple servers or use caching, which improves performance and scalability[23].

Object-Relational Mapping (ORM) in Web Services

Object-Relational Mapping (ORM)

Object-Relational Mapping (ORM) is a programming technique used to connect object-oriented applications with relational databases. Essentially, it acts as a bridge between two different systems[24]: The object-oriented paradigm used in modern programming languages, like Java, Python, or C#, and the relational model used by databases, like MySQL, PostgreSQL[25].

The purpose of ORM is to map database tables to classes and table rows to objects. This allows developers to work with database data using familiar object-oriented concepts rather than having to write raw SQL queries[26]. For example, instead of manually querying a user table, you can work with a User class and its instances.

Key purposes of ORM:

Abstraction

Hides the complexity of SQL and database interactions behind an object-oriented layer[26].

Productivity

Speeds up development by letting developers work in the language and paradigm they're familiar with[27].

Maintainability

Makes code easier to maintain and less error-prone by reducing boilerplate SQL code scattered throughout the application[27].

Portability

Enables switching between different database systems with minimal code changes, since the ORM handles the underlying database differences[28].

ORM Tools and Hibernate

ORM tools are libraries or frameworks that help manage the interaction between an object-oriented application and a relational database[29]. Instead of writing raw SQL queries, developers use these tools to perform database operations through high-level, object-oriented code. The ORM handles the mapping between classes and database tables, making the process smoother and less error-prone[30].

One of the most widely used ORM tools is **Hibernate**, a powerful framework for Java applications.

Key features of Hibernate:

Automatic mapping:

Hibernate maps Java classes to database tables and Java objects to table rows, handling the translation between the two[31].

Database independence:

With Hibernate, you can switch between different relational databases (like MySQL, PostgreSQL, or Oracle) without changing much of your code[32].

Lazy loading:

Hibernate supports fetching data only when it's needed, which can improve performance[31].

Caching:

It offers multiple caching mechanisms to reduce the number of database queries and improve speed[33].

HQL (Hibernate Query Language):

A query language similar to SQL but designed to work with Hibernate's object model rather than directly with tables and columns[32].

Why use Hibernate?

Hibernate simplifies database interactions, reduces boilerplate code, and integrates well with the Java ecosystem. It handles complex tasks like relationships (one-to-many, many-to-many), cascading, and transaction management, making it easier for developers to focus on business logic rather than low-level database details[27].

How ORM Tools Simplify Database Interactions within OOP

ORM (Object-Relational Mapping) tools make it much easier to connect object-oriented applications with relational databases by acting as an intermediary layer[24]. Instead of forcing developers to constantly switch between two different paradigms, object-oriented code and relational data. ORM tools let them focus on objects while handling the messy database details behind the scenes.

Seamless object–table mapping:

ORM tools automatically map classes to database tables and object instances to rows. This means you can work directly with objects and their properties rather than writing SQL queries and manually interpreting query results[34].

Automated data operations:

Basic tasks like saving, updating, deleting, or retrieving data are wrapped into high-level methods (often just `save()`, `update()`, or `find()`). This removes the need for repetitive SQL code and reduces the chance of errors[26].

Relationship management:

ORMs handle complex relationships (like one-to-many, many-to-many) through object references and collections, which is far more intuitive for developers than managing foreign keys and join tables manually[35].

Database abstraction:

Since ORM tools generate the SQL behind the scenes, your application code becomes less dependent on the specifics of any one database system, improving portability and flexibility[28].

Cleaner, maintainable code:

By keeping database access logic separate and standardised, ORM tools help reduce boilerplate code, making the overall codebase easier to read, maintain, and extend[27].

In essence, ORM tools let developers focus on designing and working with meaningful objects and their behaviours, while the tool itself handles the translation to and from the relational database world.

Advantages and Limitations of ORM

Reduced Boilerplate Code

ORM tools automate repetitive tasks like writing SQL queries for basic CRUD (Create, Read, Update, Delete) operations, managing database connections, and mapping result sets to objects. Instead of manually handling these low-level details, developers can use simple method calls provided by the ORM. This reduces the amount of boilerplate code throughout the application, leading to cleaner, more focused logic.

Improved Maintainability

By centralising and standardising how the application interacts with the database, ORM tools make the codebase easier to maintain. Developers no longer need to track and update raw SQL queries across multiple parts of the system — changes to the database schema (like renaming a column) can often be handled in one place, such as the mapped class. This reduces the risk of introducing errors and makes it simpler to update or refactor the application over time.

Abstraction from SQL

ORM tools provide a layer of abstraction over raw SQL, allowing developers to work with objects, methods, and high-level query APIs rather than writing database-specific SQL statements. This not only simplifies but also improves portability, the same

application code can often work with different database systems with minimal adjustments, since the ORM handles database-specific differences internally.

Limitations of Using ORM Tools

Performance Issues

While ORM tools improve developer productivity, they can introduce performance overhead. Since ORMs generate SQL automatically, the queries they produce may not be as efficient as hand-written, optimised SQL. For applications with heavy database use or time-critical operations, this overhead can lead to slower response times, especially if the ORM isn't carefully tuned.

Complexity with Large Datasets

ORMs are designed to work smoothly with object graphs and small-to-moderate datasets, but they can struggle when handling very large datasets. For example, fetching thousands of records into memory as objects can consume significant resources and cause memory bloat. Developers need to be cautious and apply strategies like pagination, lazy loading, or batching to avoid overwhelming the system.

Limitations in Query Optimisation

While ORM tools often provide their own query languages (like Hibernate's HQL), they don't always allow fine-grained control over the generated SQL. Complex or highly optimised queries, such as those requiring database-specific functions, advanced joins, or stored procedures, can be difficult or awkward to express using the ORM's abstractions. In such cases, developers may need to drop down to raw SQL or native queries, which partially defeats the purpose of using the ORM.

Spring Boot and OOP

Integration of OOP Concepts in Spring Boot

Spring Boot, as an extension of the Spring Framework, applies object-oriented programming (OOP) principles to help developers build software that is maintainable, modular, and reusable[36].

Maintainability

Spring Boot encourages a layered architecture, where responsibilities are divided between components like controllers, services, and repositories[37]. Each layer

encapsulates its own logic, while repositories abstract away access details. This separation ensures that changes within one component do not affect the entire system. Abstraction, achieved through interfaces and abstracted classes, allows developers to interact with the necessary features without worrying about how it is implemented, making the system easier to maintain and evolve over time[2].

Modularity

A cornerstone of Spring Boot is its use of dependency injection (DI), a design pattern in OOP principles[38]. It allows components to depend on abstractions rather than concrete implementations, promoting loose coupling. For example, a service can be injected with any class that implements a specific interface, making it easy to swap components in and out. This promotes modularity, allowing each component to be developed, tested, and replaced independently, improving the system's flexibility[38].

Reusability

Spring Boot's design relies on inheritance and polymorphism to maximise code reuse[8]. Many Spring-provided base classes (like `JpaRepository`) can be extended to gain powerful built-in functionality without rewriting code. Polymorphism allows different implementations of an interface to be swapped in without changing how they're used in the broader system. This makes it easy to create general-purpose components that can be reused across projects or adapted to new requirements.

Design Patterns in Spring Boot

Dependency Injection

Dependency Injection is a technique where an object's dependencies are provided from the outside rather than created within the object itself.

Alignment with OOP Principles:

Encapsulation: DI helps maintain clean class boundaries by keeping the construction of dependencies separate from their usage. The class focuses solely on its own logic, not on how its dependencies are built[1].

Abstraction & Polymorphism: With DI, you can inject any object that adheres to a given interface, allowing the system to depend on abstractions rather than concrete implementations. This makes it easy to substitute different implementations (polymorphism) without changing the consuming class[3].

Modularity & Maintainability: DI reduces tight coupling between classes, making components easier to test, replace, and maintain.

Evaluation:

DI directly strengthens modularity and flexibility. It encourages the composition over inheritance approach, where behaviour is assembled from interchangeable components, improving adaptability and supporting the open-closed principle [39] being open to extension while being closed to modification.

Repository Pattern

The Repository Pattern abstracts data access by providing a dedicated layer that mediates between the domain and data mapping layers, such as databases or external services[40].

Alignment with OOP Principles:

Abstraction: The repository hides the details of data storage and retrieval, providing a simple interface for the rest of the application to use. Developers work with objects, not database queries or APIs[3].

Encapsulation: By centralising data access logic, the repository encapsulates all interactions with the data source, shielding the rest of the application from direct exposure[1].

Reusability: Repository interfaces can be reused across services or swapped with alternative implementations[10].

Separation of Concerns: It keeps business logic separate from data access logic, promoting a clear division of responsibilities.

Evaluation:

The Repository Pattern enhances maintainability and testability, as changes in the data layer, such as switching from one database to another, typically require minimal adjustments outside the repository[40]. It also encourages clean domain models, supporting the OOP emphasis on working with meaningful objects rather than technical data structures.

In modern web applications, Dependency Injection (DI) and the Repository Pattern are not just theoretical patterns, and play a practical role in shaping clean, efficient, and scalable system architectures.

Enhancing Design Quality

Loose Coupling & Modularity:

DI ensures that web application components are loosely coupled. Instead of hardcoding dependencies, the framework injects them at runtime. This modular structure allows developers to swap components, such as replacing a mock service for testing.

Separation of Concerns:

The Repository Pattern divides responsibilities between components: controllers handle requests, services manage business logic, and repositories manage data access. This clear separation keeps the code organised and adheres to the single responsibility principle, making the system easier to understand and maintain.

Testability:

Both patterns significantly improve testability. With DI, you can inject mock or stub objects during unit testing, while repositories allow you to test business logic without needing to interact with the real database. This leads to faster development cycles and more reliable applications.

Improving Efficiency

Optimised Resource Management:

DI frameworks efficiently manage the lifecycle of components, reducing the overhead of repeatedly creating and destroying objects. Singleton services, pooled connections, and cached repositories help conserve system resources.

Database Access Efficiency:

The Repository Pattern centralises data access logic, making it easier to apply consistent optimisations like query tuning, caching, or batching across the application. Without this central layer, developers might duplicate inefficient queries across multiple services, harming performance.

Scalability:

Because the system is modular and loosely coupled, scaling or extending the application (for example, by adding new services or switching to a distributed

database) can often be done with minimal disruption. Efficient design leads to better runtime performance and easier scaling.

Best Practices

Apply Domain-Driven Design (DDD)

Define clear domain models

Focus on modelling the *core business concepts* (entities, value objects, aggregates) rather than just database tables. This keeps the code aligned with real-world logic, improving clarity and maintainability[41].

Example: Instead of scattering account-related logic across multiple services, encapsulate it within an Account aggregate with clear boundaries.

Use repositories and services thoughtfully:

Repositories should only handle data access, domain services handle business rules. Developers should avoid implementing all logic into controllers or repositories[40].

Implement Proper Transaction Management

Use @Transactional carefully

Annotate service methods that require atomic operations with Spring's `@Transactional` to ensure consistency. Keep transactions *short and focused* to reduce lock contention and improve performance[42].

Understand ORM cascading and flushing

Know how your ORM (like Hibernate) handles cascading updates and when it flushes changes to the database. Improper use can lead to unexpected writes or performance slowdowns[25].

Avoid long-running transactions

Don't hold database transactions open across multiple HTTP requests or long computations, these risks database locks and resource contention[42].

Design a Modular Architecture

- **Structure the application by feature, not layer:**

Instead of separating all controllers, services, and repositories into global packages, organise modules by business domain (e.g., user, order, payment). This improves cohesion and makes scaling or refactoring easier[37].

- **Leverage Spring Boot's auto-configuration and profiles:**

Use profiles (@Profile) to configure different behaviours for development, testing, and production environments, ensuring the app scales smoothly[43].

Use dependency injection effectively:

Inject abstractions (interfaces) rather than concrete classes to keep components loosely coupled and easily replaceable or extendable[38].

Optimise ORM Usage

- **Control eager vs. lazy loading:**

Be deliberate in choosing when to load related entities — eager loading can over-fetch data, while lazy loading can cause N+1 query issues if misused[31].

- **Use pagination and batching:**

When dealing with large datasets, always page results and batch updates to prevent memory exhaustion and reduce database load[44].

- **Monitor generated SQL:**

Use tools like Hibernate's SQL logging to check what queries are actually executed and optimise where needed[24].

Challenges and Future Perspectives

Common Implementation Challenges

Performance Overhead

Explanation:

DI frameworks add a layer of abstraction that can introduce runtime overhead, especially during application startup when the container creates and wires many objects. This can slightly slow down application initialisation, particularly in large projects with many beans and complex configurations[45].

Example:

A Spring Boot application with hundreds of services and components may experience noticeably longer startup times as the DI container resolves all dependencies and performs auto-configuration.

Lazy Loading Issues

Explanation:

Lazy loading, often used in ORM repositories (like Hibernate), delays loading related data until it's explicitly accessed. While this can improve performance by avoiding unnecessary database queries, it can also cause problems like the N+1 select issue — where fetching a list of entities leads to multiple small, inefficient queries instead of a single, optimised join[31].

Example:

When retrieving a list of Order entities, if each order's customer field is lazily loaded, accessing those customer details in a loop may trigger a separate query for each order, causing performance bottlenecks.

Data Consistency

Explanation:

With DI and repository-based designs, multiple services may operate on shared data sources concurrently. Without proper transaction management, you can encounter issues like partial updates or inconsistent reads, especially in multi-threaded or distributed environments[46].

Example:

Two services might update the same Account entity at the same time — if transactions aren't properly configured, one update might overwrite the other or leave the data in an inconsistent state, leading to application errors or data corruption.

Emerging Trends

Microservices

Impact on OOP:

Microservices break large applications into smaller, independently deployable services, each prioritising a specific business capability. While OOP still plays a role within each microservice, the emphasis shifts towards service boundaries, APIs, and inter-service communication rather than large, tightly coupled object models[47].

Impact on ORM:

Instead of a single ORM layer managing a shared database, each microservice typically owns its own database. This reduces the scope of ORM to smaller, bounded contexts but also reduces the likelihood of using complex cross-entity relationships, since relationships across services are now handled at the API level, not in the ORM[48].

Cloud-Native Development

Impact on OOP:

Cloud-native systems are designed for scalability, resilience, and rapid deployment, often using containerisation, orchestration (like Kubernetes), and serverless functions. This environment encourages a more modular, lightweight OOP design, where components must be loosely coupled and stateless to support scaling and failover[49].

Impact on ORM:

Cloud-native systems often prioritise lightweight, stateless interactions and distributed data management, which can clash with traditional ORM assumptions about stateful, tightly coupled object graphs. ORM usage might be reduced in favour of lightweight data access libraries or used selectively within services where object mapping still makes sense[50].

GraphQL APIs

Impact on OOP:

GraphQL changes how client-server interactions are designed by allowing clients to query the exact data they need, reducing over-fetching. This pushes developers to think carefully about designing flexible, well-structured domain models that can be efficiently exposed through GraphQL schemas[51].

Impact on ORM:

With GraphQL, the flexibility of queries can overload the ORM. For example, if a client requests nested data structures, the ORM needs to handle dynamic fetching efficiently without falling into N+1 query traps. As a result, developers may turn to optimised data loaders, direct SQL, or hybrid approaches (mixing ORM and non-ORM access) to meet performance demands[52].

How These Trends Might Evolve the Use of ORM

Smaller Scope, Focused Use:

ORMs will likely be used more selectively within individual microservices or bounded contexts rather than as a system-wide solution[49].

Increased Hybrid Approaches:

Developers may combine ORM with other tools like native queries, CQRS patterns, or NoSQL access to optimise performance and fit the specific needs of cloud-native and GraphQL-based architectures[52].

Shift Toward API-Centric Thinking:

As architectures become more API driven, the focus may shift away from rich, internal object models toward external-facing contracts and Data Transfer Objects (DTOs), reducing the reliance on heavy ORM-managed domain models[31].

Recommendations

Performance Tuning

Understand Fetch Strategies:

Be deliberate in choosing between *lazy* and *eager* loading. Use lazy loading for optional or infrequently accessed relationships, but switch to eager loading when related data will always be needed. This helps avoid the *N+1 query problem*[31].

Example: Fetching a list of orders should eagerly load customer details if every order page requires them.

Optimise Batching:

Configure the ORM, such as Hibernate's `batch_size`, to batch insert, update, or delete operations, reducing the number of round-trips to the database[32].

Profile and Monitor Queries:

Regularly review the SQL generated by the ORM using logging or profiling tools. Look for inefficient queries, missing indexes, or redundant joins[27].

Strategic Caching

Leverage First-Level and Second-Level Caches:

Use the ORM's built-in first-level cache to avoid redundant queries within a transaction. For broader reuse, implement second-level caching to store frequently accessed entities across sessions[53].

Cache Carefully:

Cache only data that is read-heavy and not frequently updated. Always consider cache invalidation strategies to prevent stale data issues[54].

Query Optimisation

Write Custom Queries When Needed:

ORMs are great for standard CRUD operations, but for complex queries (e.g., analytics, reporting), it's often better to write custom SQL or use native queries to ensure efficiency[26].

Use Projections and DTOs:

Avoid loading entire entities when you only need a subset of fields. Use lightweight DTOs or projections to fetch just the required data[55].

Avoid Over-Complicated Mappings:

Keep entity mappings straightforward. Avoid unnecessary or overly complex relationships that can complicate queries and hurt performance[24].

Transaction and Connection Management

Keep Transactions Short:

Ensure database transactions are kept as short as possible to avoid locking resources and holding up concurrent operations[42].

Architectural Considerations

Apply Command Query Responsibility Segregation (CQRS):

Separate read and write models, using ORM for transactional operations and optimised, possibly non-ORM, approaches (like direct SQL or caching layers) for complex reads[56].

Modularise Data Access:

Organise repositories and data access layers around clear, bounded contexts aligned with domain-driven design to keep ORM mappings focused and manageable[41].

Conclusion

In summary, the principles of Object-Oriented Programming (OOP), form the conceptual backbone of modern software engineering, enabling developers to design systems that are modular, maintainable, and scalable. When these principles are integrated into frameworks like Spring Boot, they provide not only structural clarity but also practical tools like dependency injection and the repository pattern, which directly enhance design quality, testability, and efficiency.

Web services and RESTful architectures, underpinned by these OOP concepts, further expand the ability to build interoperable, loosely coupled, and scalable systems that can communicate across diverse platforms. Meanwhile, the adoption of Object-Relational Mapping (ORM) tools such as Hibernate simplifies the traditionally complex task of bridging object-oriented code with relational databases, reducing boilerplate code, improving maintainability, and abstracting away low-level SQL operations. However, as explored, ORM comes with its own set of challenges, particularly around performance, large dataset handling, and complex queries, that require thoughtful tuning and architectural decisions.

Looking forward, emerging trends like microservices, cloud-native development, and GraphQL APIs are reshaping how OOP and ORM are applied in practice. They encourage a shift toward smaller, more focused object models, hybrid data access strategies, and an increasing emphasis on API- and contract-driven design over large, monolithic domain models.

Ultimately, mastering these principles and tools, while staying adaptable to evolving technologies and architectural patterns, equips software engineers to build systems that are not only technically sound but also resilient and future-ready. Thoughtful application of best practices, combined with ongoing awareness of new trends, ensures that developers can continue to deliver high-quality, performant, and maintainable software in an ever-changing landscape.

References

- [1] A. Snyder, 'Encapsulation and Inheritance in Object-Oriented Programming Languages', *ACM SIGPLAN Notices*, vol. 21, pp. 38–45, Jan. 1986, doi: 10.1145/960112.28702.
 - [2] N. M. Hassan, 'CONSTRUCTORS AND DESTRUCTUR IN OBJECT ORIENTED PROGRAMMING', *Journal of engineering and sustainable development (Online)*, vol. 21, no. 4, 2017.
 - [3] E. Tempero, 'How Fields are Used in Java: An Empirical Study', in *2009 Australian Software Engineering Conference*, 2009, pp. 91–100. doi: 10.1109/ASWEC.2009.19.
 - [4] A. Snyder, 'Encapsulation and inheritance in object-oriented programming languages', *Conference on Object-Oriented Programming Systems, Languages and Applications*, vol. 21, no. 11, pp. 38–45, 1986, doi: 10.1145/960112.28702.
 - [5] S. Denier and Y.-G. Guéhéneuc, 'Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies', in *2008 16th IEEE International Conference on Program Comprehension*, 2008, pp. 143–152. doi: 10.1109/ICPC.2008.37.
 - [6] R. Biddle, E. Tempero, and K. J. Klee, 'Explaining inheritance: a code reusability perspective', New York, NY, USA: ACM, 1996, pp. 217–221. doi: 10.1145/236452.236543.
 - [7] R. J. Wirfs-Brock, 'Designing Extensible Classes', *IEEE Softw*, vol. 24, no. 5, pp. 15–17, Sep. 2007, doi: 10.1109/MS.2007.137.
 - [8] H. Ali, 'Exploring Polymorphism: Flexibility and Code Reusability in Object-Oriented Programming', *Passer journal of basic and applied sciences (Online)*, vol. 6, no. Special Issue, 2024, doi: 10.24271/psr.2024.189667.
 - [9] M. Hammer, R. Maschotta, and A. Zimmermann, 'Extended Polymorphism Semantics for fUML Models', in *SEAA, IEEE*, 2024, pp. 142–151. doi: 10.1109/SEAA64295.2024.00030.
 - [10] I. Choi and M. V Mannino, 'Graph interpretation of methods: a unifying framework for polymorphism in object-oriented programming', *OOPS messenger*, vol. 2, no. 1, pp. 38–54, 1991, doi: 10.1145/122438.122441.
 - [11] A. Beugnard, 'Method overloading and overriding cause encapsulation flaw', in *Proceedings of the 2006 ACM symposium on Applied computing*, New York, NY, USA: ACM, Apr. 2006, pp. 1424–1428. doi: 10.1145/1141277.1141608.
 - [12] R. Bhatti, E. Bertino, A. Ghafoor, and J. B. D. Joshi, 'XML-based specification for Web services document security', *Computer (Long Beach, Calif.)*, vol. 37, no. 4, pp. 41–49, 2004, doi: 10.1109/MC.2004.1297300.
-

- [13] E. A. Lile, 'Client/server architecture: A brief overview', *Journal of Systems Management*, vol. 44, no. 12, p. 26, Dec. 1993, [Online]. Available: <https://www.proquest.com/scholarly-journals/client-server-architecture-brief-overview/docview/199816102/se-2?accountid=130472>
 - [14] A. Tsalgatidou and T. Pilioura, 'An overview of standards and related technology in Web Services', *Distributed and parallel databases : an international journal*, vol. 12, no. 2–3, pp. 135–162, 2002, doi: 10.1023/A:1016599017660.
 - [15] H. R. M. Nezhad, B. Benatallah, F. Casati, and F. Toumani, 'Web services interoperability specifications', *Computer (Long Beach, Calif.)*, vol. 39, no. 5, pp. 24–32, 2006, doi: 10.1109/MC.2006.181.
 - [16] A. Molina, D. Vargas, and A. Rodas, 'Implementation of A Data-Acquisition System and Its Cloud-Based Registration Using the Unified Architecture of Open Platform Communications', *Engineering proceedings*, vol. 47, no. 1, p. 20, 2023, doi: 10.3390/engproc2023047020.
 - [17] P. T. Eugster and R. Guerraoui, 'Distributed programming with typed events', *IEEE Softw*, vol. 21, no. 2, pp. 56–64, 2004, doi: 10.1109/MS.2004.1270763.
 - [18] R. Verma and A. Srivastava, 'A dynamic service description for mobile environments', *Computing*, vol. 100, no. 12, pp. 1221–1249, 2018, doi: 10.1007/s00607-018-0611-z.
 - [19] N. Jabalia, H. Bansal, V. Rahul, N. Chaudhary, A. Grover, and J. Narayan, 'Role and utilization of web services in bioinformatics', *J Nat Sci Biol Med*, vol. 2, no. 3, p. 90, 2011.
 - [20] S.-P. Ma, M.-J. Hsu, H.-J. Chen, and C.-J. Lin, 'RESTful API Analysis, Recommendation, and Client Code Retrieval', *Electronics (Basel)*, vol. 12, no. 5, p. 1252, 2023, doi: 10.3390/electronics12051252.
 - [21] A. Chourmouziadis, M. Charalambides, and G. Pavlou, 'On the Performance and Scalability of Web Services for Monitoring MPLS-based Networks', *Journal of network and systems management*, vol. 17, no. 1–2, pp. 105–136, 2009, doi: 10.1007/s10922-009-9125-5.
 - [22] S. Lee, J.-Y. Jo, and Y. Kim, 'Authentication system for stateless RESTful Web service', *Journal of computational methods in sciences and engineering*, vol. 17, no. 1_suppl, pp. S21–S34, 2017, doi: 10.3233/JCM-160677.
 - [23] A. Shieh, A. C. Myers, and E. G. Sirer, 'A stateless approach to connection-oriented protocols', *ACM transactions on computer systems*, vol. 26, no. 3, pp. 1–50, 2008, doi: 10.1145/1394441.1394444.
 - [24] R. Calderón-Moreno and D. Arenas-Seleey, 'Mapping objects through a data motor NoSQL case study: framework for web applications development',
-

Ingeniería, investigación y desarrollo, vol. 16, no. 1, pp. 61–71, 2016, doi: 10.19053/1900771X.4303.

- [25] Q. Zhang, 'Object-oriented database systems in manufacturing: selection and applications', *Industrial management + data systems*, vol. 101, no. 3, pp. 97–105, 2001, doi: 10.1108/02635570110386553.
 - [26] C. Russell, 'Bridging the Object-Relational Divide', *ACM queue*, vol. 6, no. 3, p. 18, 2008.
 - [27] D. Zmaranda, L.-L. Pop-Fele, C. Gyorödi, R. Gyorödi, and G. Pecherle, 'Performance Comparison of CRUD Methods using NET Object Relational Mappers: A Case Study', *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 1, 2020, doi: 10.14569/IJACSA.2020.0110107.
 - [28] N. Chohan, C. Bunch, C. Krintz, and N. Canumalla, 'Cloud Platform Datastore Support', *J Grid Comput*, vol. 11, no. 1, pp. 63–81, 2013, doi: 10.1007/s10723-012-9238-z.
 - [29] D. Colley, C. Stanier, and M. Asaduzzaman, 'Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks', *Journal of database management*, vol. 31, no. 4, pp. 1–23, 2020, doi: 10.4018/JDM.2020100101.
 - [30] B. Alshemaimri, R. Elmasri, T. Alsahfi, and M. Almotairi, 'A survey of problematic database code fragments in software systems', *Engineering reports (Hoboken, N.J.)*, vol. 3, no. 10, p. n/a, 2021, doi: 10.1002/eng2.12441.
 - [31] S. M. Bhatti, Z. H. Abro, and F. R. Abro, 'Performance Evaluation of Java Based Object Relational Mapping Tools', *Mehran University research journal of engineering and technology*, vol. 32, no. 2, pp. 159–166, 2013.
 - [32] A. M. Bonteanu and C. Tudose, 'Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA', *Applied sciences*, vol. 14, no. 7, p. 2743, 2024, doi: 10.3390/app14072743.
 - [33] D. A. Alvarez-Eraso and F. Arango-Isaza, 'Hibernate and spring - An analysis of maintainability against performance/Hibernate y spring - un análisis mantenibilidad contra desempeño', *Revista Facultad de Ingeniería*, no. 80, p. 97, 2016, doi: 10.17533/udea.redin.n80a11.
 - [34] D. Colley, C. Stanier, and M. Asaduzzaman, 'Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks', *Journal of database management*, vol. 31, no. 4, pp. 1–23, 2020, doi: 10.4018/JDM.2020100101.
-

- [35] G. Vial, 'Lessons in Persisting Object Data Using Object-Relational Mapping', *IEEE Softw*, vol. 36, no. 6, pp. 43–52, 2019, doi: 10.1109/MS.2018.227105428.
 - [36] Z. H. Zhang, J. J. Zhang, and X. H. Pu, 'Design and Implement of Teaching Resources Management Network Platform Based on MVC', *Applied Mechanics and Materials*, vol. 631–632, pp. 999–1002, 2014, doi: 10.4028/www.scientific.net/AMM.631-632.999.
 - [37] K. Krekora, 'Performance analysis of .Net and Spring Boot microservices on Microsoft Azure', *Journal of Computer Sciences Institute*, vol. 33, pp. 258–263, 2024, doi: 10.35784/jcsi.6268.
 - [38] Z. A. Kocsis and J. Swan, 'Dependency Injection for Programming by Optimization', 2017, *Cornell University Library*, *arXiv.org*, *Ithaca*.
 - [39] C. Xu and J. Hughes, 'Realizing the open-closed principle', in *Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05)*, 2005, pp. 274–279. doi: 10.1109/ICIS.2005.107.
 - [40] J. Kanjilal, 'How to implement the Repository design pattern in C#', *InfoWorld.com*, 2016.
 - [41] S. Schimpf, K. Wussow, and M. Zeh, 'Back to the roots: imagining the application of Bauhaus methodologies in design thinking', *International journal of technology management*, vol. 95, no. 3–4, pp. 335–360, 2024, doi: 10.1504/IJTM.2024.138847.
 - [42] 'Spring 2015 Special Issue on Content Management', *IEEE Trans Prof Commun*, vol. 56, no. 3, pp. 264–268, Sep. 2013, doi: 10.1109/TPC.2013.2278755.
 - [43] R. Książek and B. Pańczyk, 'Comparative analysis of selected programming frameworks of Java-based web applications', *Journal of Computer Sciences Institute*, vol. 23, pp. 66–70, 2022, doi: 10.35784/jcsi.2787.
 - [44] S. Azizian, E. Rastegari, and H. Bagheri, 'Leveraging Machine Learning for Optimal Object-Relational Database Mapping in Software Systems', in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, New York, NY, USA: ACM, Jul. 2024, pp. 94–102. doi: 10.1145/3664646.3664769.
 - [45] K. Guntupally, R. Devarakonda, and K. Kehoe, 'Spring Boot based REST API to Improve Data Quality Report Generation for Big Scientific Data: ARM Data Center Example', in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 5328–5329. doi: 10.1109/BigData.2018.8621924.
 - [46] J. Ahmed, A. Karpenko, O. Tarasyuk, A. Gorbenko, and A. Sheikh-Akbari, 'Consistency issue and related trade-offs in distributed replicated systems and
-

databases: a review', *Radioelektronni i komp'uterni sistemi (Online)*, no. 2, pp. 171–179, 2023, doi: 10.32620/reks.2023.2.14.

- [47] V. Velepucha and P. Flores, 'A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges', *IEEE Access*, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.
 - [48] I. Omar, E. K. E. H. Mohamed, and Z. Anass, 'Microhooks: A Novel Framework to Streamline the Development of Microservices', *Computers*, vol. 14, no. 4, p. 139, 2025, doi: <https://doi.org/10.3390/computers14040139>.
 - [49] M. Chaturvedi, S. J. Ram, S. S. Shingade, and S. R. Swamy, 'From Monolith to Microservices: A Systematic Literature Survey', in *ICDDS*, IEEE, 2024, pp. 1–6. doi: 10.1109/ICDDS62937.2024.10910708.
 - [50] B. Nascimento, R. Santos, J. Henriques, M. V Bernardo, and F. Caldeira, 'Availability, Scalability, and Security in the Migration from Container-Based to Cloud-Native Applications', *Computers*, vol. 13, no. 8, p. 192, 2024, doi: <https://doi.org/10.3390/computers13080192>.
 - [51] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, 'GraphQL: A Systematic Mapping Study', *ACM Comput Surv*, vol. 55, no. 10, pp. 1–35, 2023, doi: 10.1145/3561818.
 - [52] H. Li, O. Hartig, R. Armiento, and P. Lambrix, 'Ontology-based GraphQL server generation for data access and data integration', *Semant Web*, vol. 15, no. 5, p. 1639, 2024, doi: 10.3233/sw-233550.
 - [53] S. Kuai, Y. Hu, X. Zhao, D. Qin, W. Li, and X. Li, 'A Novel Efficient Query Strategy on Hibernate', New York, NY, USA: ACM, 2019, pp. 105–108. doi: 10.1145/3316615.3316717.
 - [54] G. Cao, 'Proactive power-aware cache management for mobile computing systems', *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 608–621, 2002, doi: 10.1109/TC.2002.1009147.
 - [55] E. L. Romanov, G. V Troshina, and S. A. Menzhulin, 'Client-server application framework based on an object-oriented network model', *IOP Conf Ser Mater Sci Eng*, vol. 919, no. 5, p. 52014, 2020, doi: 10.1088/1757-899X/919/5/052014.
 - [56] S. Diakov, T. Zubrei, and A. Samoidiuk, 'APPLICATION OF EVENT SOURCING AND CQRS PATTERNS IN DISTRIBUTED SYSTEMS', *Adaptivni sistemi avtomatičnogo upravlennâ (Online)*, vol. 1, no. 34, 2019, doi: 10.20535/1560-8956.1.2019.178224.
-

Critical analysis of the relationship between OOP and Web Services_RyanEvans_2420520

GRADEMARK REPORT

FINAL GRADE

68/100

GENERAL COMMENTS

A good piece of work overall. As a general comment, you have a lot of headings followed by relatively short sections - this is an approach you should try to move away from somewhat, as it is causing your work to have a less robust narrative flow than it may otherwise have. For your upcoming project, try to write in fuller paragraphs.

A good use of sources, lots of academically credible sources used.

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

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28
