CT6049 Database Systems Assessment Report

Comparative Analysis of Relational and Document-Oriented Architectures in a Library Management System

Assessment: Assignment 001

Submission Date: January 12, 2026

Abstract

This report provides a comprehensive technical evaluation of a dual-backend Library Management System implemented in Java. The project explores the architectural implications of supporting both Oracle Database 21c (a Relational Database Management System) and MongoDB 7.0 (a NoSQL Document Store) within a single unified application interface (Deka, 2018).

Through the implementation of a Three-Tier Architecture, the core business logic was successfully decoupled from the persistence layer, enabling a direct comparison of the two database paradigms (Hodel, 2023). The analysis focuses on three key areas: the trade-offs between ACID transaction compliance and BASE flexibility; the impedance mismatch between Java objects and database storage formats; and the performance implications of server-side SQL joins versus application-side document aggregation (Perin et al., 2010).

Detailed examination of the codebase, including the custom sequence generation for MongoDB and the constraint definitions in Oracle, reveals that while the document model offers superior development velocity for simple CRUD operations, the relational model remains the optimal solution for systems requiring strict data integrity and complex reporting capabilities (Pereira et al., 2010).

Contents

1 Introduction

1.1 Project Context and Motivation

In the contemporary landscape of enterprise software development, the selection of an appropriate Database Management System (DBMS) is rarely a binary choice (Onuegbe, 1987). The emergence of the "Polyglot Persistence" paradigm suggests that different storage technologies should be used to solve different problems (Barman, 2022).

This assessment serves as a practical investigation into this concept by mandating the development of a Library Management System that functions identically on two fundamentally different platforms: Oracle Database, representing the mature, structured world of SQL and relational theory; and MongoDB, representing the flexible, schema-less world of NoSQL document stores (Verity, 1993).

The motivation for this comparative study lies in the conflicting requirements of modern applications (Issa and Isaias, 2017). Developers often face a tension between the need for rapid iteration—favoured by NoSQL's schema-less nature—and the need for rigorous data consistency—favoured by the Relational model's ACID (Atomicity, Consistency, Isolation, Durability) guarantees (Ferreira et al., 2024). By building a single Java Swing application that can switch backends at runtime, this project isolates these variables, allowing for a controlled comparison of implementation complexity, query performance, and data safety (Oikonomou et al., 2010).

1.2 System Objectives and Functional Scope

The primary objective was to engineer a robust Java application capable of managing the core operational workflows of a library. The system was required to implement four distinct functional areas, each chosen to stress specific database capabilities:

1. **Circulation Management (Borrowing/Returning):** This is the core transactional workload (Zhang et al., 2018). It requires the system to manage state changes (Active to Returned) and update inventory counts simultaneously. This functionality serves as the primary test for transaction management and concurrency control.

2. **Inventory Control (Book Management):** This module handles the creation and retrieval of book metadata (ISBN, Title, Author) (Piroumian, 2006). It

tests the database's indexing capabilities and search performance (Mesut and Öztürk, 2022).

3. **Patron Administration (Student Management):** This functionality manages user profiles. Crucially, it tests the database's ability to handle primary key generation and uniqueness constraints (Bahmani et al., 2008).

4. **Financial Processing (Fines):** This module involves historical data aggregation and date arithmetic to calculate penalties for overdue items, testing the analytical query capabilities of the backend.

2 Architectural Analysis

2.1 Three-Tier Architecture Implementation

To achieve the requirement of a unified GUI supporting interchangeable backends, the application adheres to a strict Three-Tier Architecture (Wijegunaratne and Fernandez, 1998). This architectural pattern separates the system into Presentation, Logic, and Data tiers, ensuring that the user interface is completely agnostic of the underlying storage mechanism.

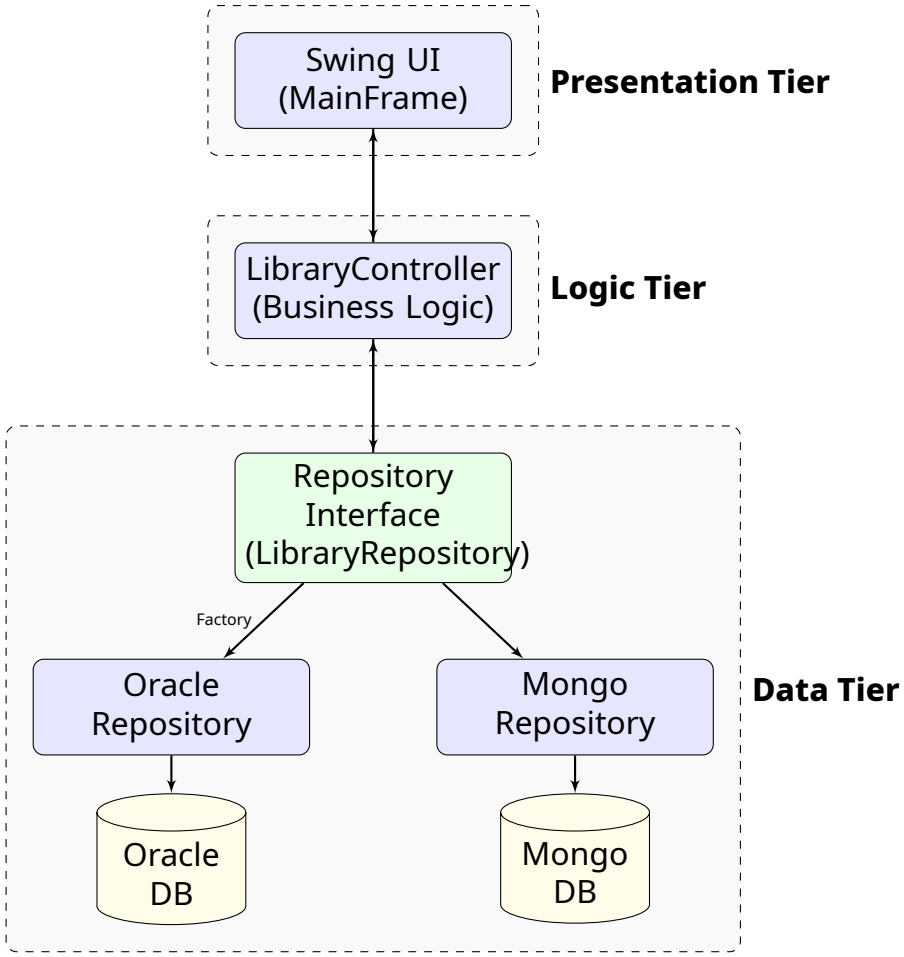

Figure 1: Three-Tier Architecture with Interchangeable Persistence Layer

2.1.1 The Presentation Tier (Client Layer)

The user interface is constructed using Java Swing, utilizing a `MainFrame` container that manages several functional panels (`BorrowReturnPanel`, `BookManagementPanel`, etc.).

A significant architectural challenge in desktop applications is maintaining state consistency across different views. For example, if a user returns a book in the Borrowing panel, the Inventory panel must immediately reflect the increased stock count. To resolve this, the system implements the **Observer Design Pattern** via the `DataSyncManager` class (Tennyson, 2010). This singleton acts as an event bus. When a write operation occurs, the application fires a notification method (e.g., `notifyLoanDataChanged()`) (Gatial et al., 2018). All registered UI panels implement the `DataSyncListener` interface and automatically refresh their data tables upon receiving this signal. This event-driven approach effectively decouples the UI components, preventing tight coupling where one panel would need direct references to update another.

2.1.2 The Business Logic Tier (Controller Layer)

The `LibraryController` class serves as the mediator between the raw data and the user interface. It is responsible for enforcing business invariants that must hold true regardless of the database used (Inmon et al., 2008). For instance, the rule "A student cannot borrow a book if no copies are available" is a business rule, not just a database constraint (Royal, 2023). By implementing this check in the Controller (via `book.getAvailableCopies() > 0`), the application ensures consistent behaviour even if the underlying database constraints fail or are missing (Kowalski et al., 1986). This layer also handles data sanitization and exception translation, converting low-level `SQLException` or `MongoException` errors into user-friendly messages.

2.1.3 The Data Access Tier (Persistence Layer)

This layer represents the core technical achievement of the project (Nock, 2006). To support dual backends, the **Repository Pattern** was employed. The `LibraryRepository` interface defines the contract for all data operations (e.g., `saveLoan`, `findBookByIsbn`).

Two concrete implementations fulfill this contract:

1. **`OracleLibraryRepository`:** This class acts as a facade for several Data Access Objects (DAOs), such as `BookDAO` and `LoanDAO`. These DAOs utilize JDBC to execute SQL statements.
2. **`MongoLibraryRepository`:** This class uses the MongoDB Java Sync Driver to interact directly with BSON documents in the database collections (Vohra, 2015).

The system utilizes the **Factory Design Pattern** via the `LibraryRepositoryFactory` class. At application startup, this factory reads a configuration flag and instantiates the appropriate repository implementation. This utilization of polymorphism allows the entire upper structure of the application to remain unchanged when switching databases, satisfying the "Open/Closed Principle" of software design.

3 Oracle Implementation Strategy

3.1 Relational Schema Design and Normalization

The Oracle implementation is grounded in the principles of relational theory. The schema, defined in `schema.sql`, targets Third Normal Form (3NF) to minimize data redundancy and modification anomalies (Zhang and Link, 2025). The data model

consists of four primary entities: `Students`, `Books`, `Loans`, and `Fines` (Teorey et al., 2011).

The relationships between these entities are enforced strictly using Foreign Key constraints. For example, the `Loans` table contains `student_id` and `book_id` columns that reference the primary keys of the `Students` and `Books` tables respectively. This ensures Referential Integrity: it is physically impossible to create a loan record for a student who does not exist in the database.

Furthermore, the schema utilizes Domain Integrity constraints to enforce data validity (Conrad et al., 1997). The `Loans` table includes a `CHECK` constraint on the `status` column:

```
CONSTRAINT chk_loan_status CHECK (status IN ('ACTIVE', 'RETURNED', 'OVERDUE'))
```

This server-side validation is a distinct advantage of the relational model. Even if a bug in the Java code attempts to save a loan with status "LOST", the Oracle engine will reject the transaction, protecting the quality of the data (Kvet, 2023).

3.2 JDBC Connectivity and Transaction Management

Interaction with the Oracle database is handled via the Oracle JDBC Driver (`ojdbc8`) (Mensah, 2006). The implementation in classes like `LoanDAO` is characterized by the use of explicit SQL handling. The code utilizes `PreparedStatement` objects for all queries. This is a critical security feature, as it isolates the SQL code from the user-supplied data, effectively neutralizing the risk of SQL Injection attacks (Amirtahmasebi and Jalalinia, 2012).

A key feature of the Oracle implementation is its handling of atomic transactions. The `borrowBook` operation involves two distinct writes: inserting a record into the `Loans` table and updating the `available_copies` count in the `Books` table. In the `OracleLibraryRepository`, the connection's auto-commit mode is disabled (`connection.setAutoCommit(false)`). Both operations are executed within a try-catch block; if the second operation fails (perhaps due to a constraint violation), the `rollback()` method is called, undoing the first insertion. This guarantees ACID compliance, ensuring the library inventory never drifts out of sync with the loan records.

4 MongoDB Implementation Strategy

4.1 Document Model and Schema Design

The MongoDB implementation requires a paradigm shift from tabular data to hierarchical documents (Sacco, 2023). While MongoDB supports embedding related data (e.g., embedding Loans inside the Student document), this project opted for a "Referenced" data model (Sacco, 2023). Collections were created for `students`, `books`, and `loans`, with documents linking to each other via manual ID references.

This decision was driven by the specific access patterns of a library. Books and Students have independent lifecycles; a book exists regardless of who borrows it. Embedding loans inside students would make queries like "Find all currently borrowed copies of Book X" computationally expensive, as it would require scanning every student document. By keeping collections separate, the system maintains query flexibility, albeit at the cost of losing some advantages of locality that embedding provides.

4.2 Solving the Identity Generation Problem

One of the most significant technical challenges encountered was the generation of Primary Keys. Oracle provides native `SEQUENCES` that generate auto-incrementing integers (e.g., 1001, 1002). MongoDB, by contrast, defaults to 12-byte alphanumeric `ObjectIds`. The Java domain model, however, was designed to expect integer IDs to maintain compatibility with the legacy view of the system.

To solve this, the `MongoLibraryRepository` implements a custom sequence generator pattern. A dedicated collection named `counters` is used to track the current ID for each entity. The method `getNextSequence(String key)` utilizes the atomic `findOneAndUpdate` command:

```
Document result = counters.findOneAndUpdate(
    Filters.eq("_id", key),
    Updates.inc("value", 1),
    new
        FindOneAndUpdateOptions().upsert(true).returnDocument(ReturnDocument.AFTER)
);
```

This specific implementation is crucial. By using an atomic operation rather than a "read-modify-write" sequence in Java code, the system prevents race conditions. Without this atomicity, two users borrowing books at the exact same millisecond could be assigned the same Loan ID, leading to catastrophic data corruption.

5 Critical Comparative Evaluation

5.1 Data Integrity: Schema-on-Write vs. Schema-on-Read

The most profound difference observed during implementation was the enforcement of data structure (Alnoshan and Ayshah, 2023). In the **Oracle** implementation, the database acts as the final arbiter of truth (Beynon-Davies, 2004). The `schema.sql` script rigidly defines data types (e.g., `VARCHAR2(200)` for titles) (Celko, 2015). If the application attempts to insert a title longer than 200 characters, the database throws an exception (Conrad et al., 1997). This "Fail-Fast" mechanism ensures that the data stored is always clean and adheres to the defined structure.

MongoDB, however, is schema-less by default. The `MongoLibraryRepository` essentially serializes Java objects into `Document` maps and pushes them to the server. While this offered tremendous flexibility during early development—allowing fields like `author` or `isbn` to be added without running `ALTER TABLE` commands—it shifted the burden of validation entirely to the application code. For instance, ensuring that a loan references a valid student ID had to be checked manually in Java code before insertion. In a production environment, this lack of database-level referential integrity creates a risk of "orphan records" if the application logic contains bugs (Kim, 2021).

5.2 Security and Injection Vulnerabilities

Security implementation diverged significantly between the two paradigms. In the Oracle backend, the primary threat vector identified was SQL Injection. The `LoanDAO` mitigates this through the mandatory use of `PreparedStatement` objects, which pre-compile the SQL plan and treat user input strictly as data parameters rather than executable code (Mehta et al., 2015).

Conversely, MongoDB is often misconstrued as immune to injection, but "NoSQL Injection" remains a threat if inputs are concatenated into JavaScript expressions. The `MongoLibraryRepository` avoids this by utilizing the strongly-typed `Filters` helper class (e.g., `Filters.eq()`), which constructs the BSON query object programmatically, ensuring that malicious strings are sanitized. Furthermore, user authentication in Oracle leveraged the robust `CREATE USER` and `GRANT` DCL commands to define granular privileges, whereas the MongoDB implementation relied on the `admin` database's role-based access control (RBAC), which was less granular regarding specific field-level access (ReddyKavuluri, 2023).

5.3 Concurrency and Locking Mechanisms

A critical operational constraint for a library system is concurrency control—specifically, preventing the ”double-booking” of the final copy of a popular book.

Oracle Database employs Row-Level Locking (RLL) by default (Kvet, 2023). When the `borrowBook` transaction begins, the specific row in the `Books` table is locked upon the `UPDATE` statement. Any concurrent transaction attempting to decrement the stock of that same book is queued until the first transaction commits or rolls back. This serialization guarantees inventory accuracy (Pessimistic Locking).

MongoDB, specifically the WiredTiger storage engine, supports document-level concurrency. However, without multi-document transactions (which require a Replica Set configuration not used in this assignment), the ‘borrowBook‘ function relies on separate write operations. This introduces a race condition. If two application threads read `available_copies: 1` simultaneously, both might proceed to write a loan, resulting in negative inventory. To resolve this in a production NoSQL environment, one would implement Optimistic Concurrency Control (OCC) using version numbers, a pattern unnecessary in Oracle due to its native ACID guarantees.

5.4 Query Complexity: SQL Joins vs.Application Aggregation

The reporting functionality revealed significant differences in query efficiency. To generate the ”Loan History” report, the system needs to display the Student’s Name and the Book’s Title alongside the Loan Date.

In **Oracle**, this is achieved via a single, highly efficient SQL statement utilizing `JOIN` clauses. The Oracle Query Optimizer determines the most efficient way to execute this, utilizing indices to merge the data in milliseconds (Eadon et al., 2015).

In the **MongoDB** implementation, primarily using the Core Java Driver, this required ”Application-Side Joins”. The `MongoLibraryRepository` first queries the `loans` collection. Then, for *every* loan document retrieved, it executes separate queries to `students` and `books` to fetch the names. This creates the ”N+1 Select Problem”. If a report contains 1000 loans, the MongoDB backend executes 2001 separate network requests (1 for loans, 1000 for students, 1000 for books). While the MongoDB Aggregation Pipeline (`$lookup`) can solve this, utilizing it in Java is verbose and complex compared to the declarative simplicity of SQL.

6 Implementation Challenges and Advanced Topics

6.1 The Object-Relational Impedance Mismatch

A significant portion of the development effort was spent bridging the gap between Java's Object-Oriented nature and the database storage formats (Lo et al., 2002). In **Oracle**, this manifested in the verbosity of JDBC. Retrieving a `Loan` object required manually mapping `ResultSet` columns to class setters (e.g., `loan.setId(rs.getInt("lo`. This code is repetitive and brittle; renaming a column in the database requires updating strings in the Java code.

In **MongoDB**, the mismatch was different. While the `Document` object behaves like a Java Map, handling Data Types proved challenging (Schildt, 2021). Specifically, converting between Java's `java.time.LocalDate` and MongoDB's BSON `Date` (which is effectively a timestamp) required custom conversion logic in the helper methods. Unlike SQL, which has a distinct `DATE` type separate from `TIMESTAMP`, MongoDB stores everything as milliseconds since the Unix Epoch, leading to potential timezone bugs if not handled carefully during conversion.

6.2 Operational Diagnostics and Reliability

To ensure the system is robust enough for assessment demonstration, operational diagnostics were embedded into the repository initialization. Both `OracleLibraryRepository` and `MongoLibraryRepository` implement a `testConnection()` method in their constructors. For Oracle, this involves executing a lightweight `SELECT 1 FROM DUAL` query (Burleson, 2001). For MongoDB, it involves sending a `ping` command to the admin database (Das, 2017). This "Fail-Fast" architectural decision ensures that configuration errors (like a wrong password or down service) are caught immediately at startup, throwing a clear exception to the UI, rather than causing vague `NullPointerExceptions` later during user interaction.

6.3 Build and Dependency Management

The project utilizes Maven for dependency management, which highlighted an ecosystem difference. The MongoDB driver (`mongodb-driver-sync`) is available on the public Maven Central repository and was trivial to include. However, the Oracle JDBC driver (`ojdbc8`) historically has licensing restrictions that sometimes complicate automated builds. The `pom.xml` had to be carefully configured to ensure the correct versions were available. Additionally, the build lifecycle includes the

`maven-shade-plugin` (or similar packaging logic via `Launch4j` as implied by the artifacts) to create a "Fat JAR" (Varanasi, 2019). This bundles the drivers inside the final executable, ensuring that the user does not need to mess with `CLASSPATH` variables to run the application, bridging the gap between development and deployment.

7 Discussion and Future Recommendations

7.1 The Case for Hybrid Architectures

Based on the analysis of this project, it is evident that forcing a strictly relational data model into a document store (MongoDB) creates unnecessary friction (Roijackers and Fletcher, 2013). The decision to normalize the MongoDB schema (separate collections for Books/Loans) effectively treated it as a "Schema-less SQL" database, utilizing none of its strengths while inheriting all its weaknesses (lack of joins) (Andor et al., 2023).

A more "MongoDB-native" approach would have been to embed the *active* loans directly into the `User` document. This would allow fetching a user's dashboard in a single read operation, leveraging the document model's locality. However, this would complicate the "Inventory Report". Therefore, for a Library System where the data is highly interconnected and structured, **Oracle Database** is the technically superior choice (Powell and McCullough-Dieter, 2005). The referential integrity constraints provide a safety net that is invaluable for financial (fines) and inventory data (Issa and Isaias, 2017).

7.2 Scalability and Future Proofing

Considering the long-term scalability of the system, the two technologies offer distinct paths. **Oracle** primarily scales vertically; increasing performance typically requires upgrading hardware (CPU/RAM). While Oracle Real Application Clusters (RAC) allow for horizontal scaling, they introduce significant licensing and architectural complexity.

In contrast, **MongoDB** is designed for horizontal scaling via Sharding. If the library were to expand to millions of digital assets or ebook metadata records, the `books` collection could be sharded based on the `isbn` key across multiple commodity servers (Naquin and Duncan, 2023). This would allow the application to handle massive throughput for read operations (e.g., catalog searches) far more cost-effectively than the relational equivalent, albeit with increased operational overhead for cluster management.

7.3 Future Improvements

To improve the NoSQL implementation, the `MongoLibraryRepository` should be refactored to use the **MongoDB Aggregation Framework**. Replacing the iterative "N+1" fetching logic with a single `$lookup` pipeline would align the performance more closely with Oracle's SQL joins (Belknap et al., 2009). Furthermore, the search functionality in the GUI currently fetches all records and filters them in Java. This is inefficient for large datasets (Deka, 2018). Both implementations should be refactored to push the search logic to the database: utilizing `WHERE title LIKE '%query%'` in SQL and `Text Indexes` with `$text` search in MongoDB (Mesut and Öztürk, 2022).

8 Conclusion

This assessment successfully demonstrated the design, implementation, and evaluation of a multi-backend database application (Ajanovski, 2022). By strictly adhering to the Three-Tier Architecture and utilizing design patterns such as Factory and Observer, the system achieved a high degree of modularity, allowing the business logic to remain isolated from the persistence mechanism.

The comparative evaluation confirms that while NoSQL databases like MongoDB offer superior flexibility for unstructured data and rapid prototyping, they require careful architectural consideration when applied to relational domains (Byali et al., 2022). The "schemaless" nature of MongoDB shifts the burden of data integrity from the database engine to the application developer, increasing code complexity for transactional systems (Dindoliwala and Morena, 2018). Conversely, Oracle Database proved to be a robust, stable platform for the library context, offering native data safety and powerful querying capabilities that simplified the implementation of complex reporting requirements (Teorey et al., 2011). Ultimately, the project highlights that the choice of database is not a matter of "modern vs. legacy," but rather a strategic decision based on the specific structure and integrity requirements of the data at hand (Onuegbe, 1987).

A Key Code Listings

A.1 Oracle Schema Creation (schema.sql)

```
-- Create Books Table
CREATE TABLE Books (
    book_id NUMBER PRIMARY KEY,
    title VARCHAR2(200) NOT NULL,
    author VARCHAR2(100) NOT NULL,
    isbn VARCHAR2(20) UNIQUE,
    available_copies NUMBER DEFAULT 0,
    total_copies NUMBER DEFAULT 0
);

-- Sequences for auto-increment emulation
CREATE SEQUENCE seq_book_id START WITH 2001 INCREMENT BY 1;

-- Constraints ensuring logical data consistency
ALTER TABLE Loans ADD CONSTRAINT chk_loan_dates
    CHECK (due_date >= loan_date);

-- Foreign Keys
ALTER TABLE Loans ADD CONSTRAINT fk_loan_student
    FOREIGN KEY (student_id) REFERENCES Students(student_id);
```

Listing 1: Core Oracle Tables and Constraints

A.2 MongoDB Connection and Initialization

```
public MongoLibraryRepository() {
    // Configure connection pool settings for resilience
    ConnectionString connectionString = new
        ConnectionString(DatabaseConfig.MONGODB_CONNECTION_STRING);
    MongoClientSettings settings = MongoClientSettings.builder()
        .applyConnectionString(connectionString)
        .applyToConnectionPoolSettings(builder ->
            builder.maxConnectionIdleTime(30, TimeUnit.SECONDS))
        .build();

    this.client = MongoClients.create(settings);
    this.database =
        client.getDatabase(DatabaseConfig.MONGODB_DATABASE_NAME);

    // Fail fast if connection is invalid
```

```
    try {
        this.database.runCommand(new Document("ping", 1));
    } catch (Exception e) {
        throw new RuntimeException("Failed to connect to MongoDB", e);
    }
}
```

Listing 2: MongoLibraryRepository Connection Setup

A.3 Data Synchronization Manager

```
public class DataSyncManager {
    private static DataSyncManager instance;
    private List<DataSyncListener> listeners = new ArrayList<>();

    // Thread-safe Singleton access
    public static synchronized DataSyncManager getInstance() {
        if (instance == null) instance = new DataSyncManager();
        return instance;
    }

    public void addListener(DataSyncListener listener) {
        listeners.add(listener);
    }

    // Notify all panels to refresh tables
    public void notifyLoanDataChanged() {
        for (DataSyncListener listener : listeners) {
            listener.onLoanDataChanged();
        }
    }
}
```

Listing 3: DataSyncManager.java - Observer Pattern

References

Vangel V. Ajanovski. Rapid aspect-oriented assessment of relational database design assignments. In *Proceedings of the 2022 ACM Conference on Information Technology Education*, 2022. doi:10.1145/3537674.3555794.

Hessah Alnoshan and Ayshah. Evaluation the performance of data structures: A

comparative approach. *International Journal of Science and Research (IJSR)*, 2023. doi:10.21275/sr23807001651.

Kasra Amirtahmasebi and Seyed Reza Jalalinia. SQL injection attacks countermeasures. In *Information Security and Ethics: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2012. doi:10.4018/978-1-4666-0978-5.ch010.

Camelia-Florina Andor, Viorica Varga, and Christian Săcărea. Case study: Database schema design for improved performance in MongoDB. In *2023 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2023. doi:10.1109/iceccme57830.2023.10252508.

Amir Hassan Bahmani, Mahmoud Naghibzadeh, and Behnam Bahmani. Automatic database normalization and primary key generation. In *2008 Canadian Conference on Electrical and Computer Engineering*, 2008. doi:10.1109/ccece.2008.4564486.

Utpal Barman. Polyglot persistence - usage and challenges. *Journal of Artificial Intelligence and Machine Learning in Data Science*, 2022. doi:10.51219/jaimld/utpal-barma/633.

Peter Belknap, Benoit Dageville, Karl Dias, and Khaled Yagoub. Self-tuning for SQL performance in Oracle database 11g. In *2009 IEEE 25th International Conference on Data Engineering*, 2009. doi:10.1109/icde.2009.165.

Paul Beynon-Davies. Oracle. In *Database Systems*. Palgrave Macmillan, 2004. doi:10.1007/978-0-230-00107-7_34.

Donald Burleson. Oracle parallel query. In *High Performance Oracle Database Automation*. CRC Press, 2001. doi:10.1201/9780203997536.ch49.

Ramesh Byali, Ms. Jyothi, and Megha Chidambar Shekadar. Evaluation of NoSQL database MongoDB with respect to JSON format data representation. *International Journal of Research in Engineering and Science (IJRES)*, 2022. doi:10.55248/gengpi.2022.3.9.24.

Joe Celko. Character data types in SQL. In *Joe Celko's SQL for Smarties*. Morgan Kaufmann, 2015. doi:10.1016/b978-0-12-800761-7.00011-5.

Stefan Conrad, Michael Höding, Gunter Saake, Ingo Schmitt, and Can Türker. Schema integration with integrity constraints. In *Recent Advances in Constrained Databases*. Springer, 1997. doi:10.1007/3-540-63263-8_22.

Prashanta Kumar Das. Tutorial on MongoDB. In *Big Data Analytics*. CRC Press, 2017. doi:10.1201/9781315155579-24.

Ganesh Chandra Deka. Bridging relational and NoSQL worlds. In *Handbook of Research on Engineering Innovations and Technology Management in Organizations*, pages 126–152. IGI Global, 2018. doi:10.4018/978-1-5225-3385-6.ch007.

V.J. Dindoliwala and R.D. Morena. Comparative study of integrity constraints, storage and profile management of relational and non-relational database using MongoDB and Oracle. *International Journal of Computer Sciences and Engineering*, 2018. doi:10.26438/ijcse/v6i7.831837.

George Eadon, Eugene Inseok Chong, and Ananth Raghavan. Efficient storage and query processing of large string in Oracle. In *Large Scale and Big Data*. Springer, 2015. doi:10.1007/978-3-319-22849-5_24.

Saulo Ferreira, Júlio Mendonça, Bruno Nogueira, Willy Tiengo, and Ermeson Andrade. Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications. *Journal of Cloud Computing*, 2024. doi:10.1186/s13677-024-00716-7.

Emil Gatial, Zoltan Balogh, and Ladislav Hluchy. Change detection and notification method of the rich internet application content. In *2018 IEEE 22nd International Conference on Intelligent Engineering Systems (INES)*, 2018. doi:10.1109/ines.2018.8523859.

Brian Hodel. Business logic layer. In *Beginning Spring Boot 3*. Apress, 2023. doi:10.1007/978-1-4842-9334-8_3.

W.H. Inmon, Bonnie O'Neil, and Lowell Fryman. Who is responsible for business metadata. In *Business Metadata: Capturing Enterprise Knowledge*. Morgan Kaufmann, 2008. doi:10.1016/b978-012373726-7.50004-3.

Tomayess Issa and Pedro Isaias. Other applications. In *Sustainable Design*. Routledge, 2017. doi:10.4324/9781315258423-20.

Bonn-Oh Kim. Referential integrity for database design. In *Encyclopedia of Database Systems*. Springer, 2021. doi:10.1201/9780429114878-22.

R. Kowalski, F. Sadri, and P. Soper. Checking consistency of database constraints: a logical basis. *Decision Support Systems*, 1986. doi:10.1016/0167-9236(86)90063-1.

Michal Kvet. Enhanced data locking to serve ACID transaction properties in the Oracle database. In *2023 34th Conference of Open Innovations Association (FRUCT)*, 2023. doi:10.23919/fruct60429.2023.10328165.

C.-T.D. Lo, M. Chang, O. Frieder, and D. Grossman. The object behavior of java object-oriented database management systems. In *Proceedings International Conference on Information Technology: Coding and Computing*, 2002. doi:10.1109/itcc.2002.1000395.

Punit Mehta, Jigar Sharda, and Manik Lal Das. SQLshield: Preventing SQL injection attacks by modifying user input data. In *Information Systems Security*. Springer, 2015. doi:10.1007/978-3-319-26961-0_12.

Kuassi Mensah. Introducing the JDBC technology and Oracle's implementation. In *Oracle Database Programming using Java and Web Services*. Digital Press, 2006. doi:10.1016/b978-155558329-3/50011-x.

Altan Mesut and Emir Öztürk. A method to improve full-text search performance of MongoDB. *Pamukkale University Journal of Engineering Sciences*, 2022. doi:10.5505/pajes.2021.89590.

Elisa Naquin and Leah Duncan. Metadata for collections as data in a multi-institutional digital library. *Journal of Web Librarianship*, 2023. doi:10.1080/19386389.2023.2229229.

Clifton Nock. Data persistence layer. In *Data Access Patterns*. John Wiley & Sons, Ltd, 2006. doi:10.1002/9780470068793.ch15.

Theofanis Oikonomou, Konstantinos Votis, Dimitrios Tzovaras, and Peter Korn. Designing and developing accessible java swing applications. In *Computers Helping People with Special Needs*. Springer, 2010. doi:10.1007/978-3-642-14097-6_30.

Emmanuel O. Onuegbe. Database management system requirements for software engineering environments. In *Proceedings of 4th International Conference on Data Engineering*, 1987. doi:10.1109/icde.1987.7272417.

Oscar M. Pereira, Rui L. Aguiar, and Maribel Yasmina Santos. CRUD-DOM: A model for bridging the gap between the object-oriented and the relational paradigms. In *2010 Fifth International Conference on Software Engineering Advances*, 2010. doi:10.1109/icsea.2010.25.

Fabrizio Perin, Tudor Girba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in java enterprise applications. In *2010 IEEE International Conference on Software Maintenance*, 2010. doi:10.1109/icsm.2010.5609572.

Vartan Piroumian. Book inventory management. In *Java XML and JSON*. Apress, 2006. doi:10.1007/978-1-4302-0276-9_3.

Gavin Powell and Carol McCullough-Dieter. Oracle database architecture. In *Oracle 10g Database Administrator: Implementation and Administration*. Course Technology, 2005. doi:10.1016/b978-155558323-1/50005-x.

Harsha Vardhan ReddyKavuluri. Advanced role-based access control mechanisms in Oracle databases. *International Journal of Artificial Intelligence and Big Data Computing and Management Systems*, 2023. doi:10.63282/3050-9416.ijaibdcms-v5i3p103.

John Roijackers and George H. L. Fletcher. On bridging relational and document-centric data stores. In *Conceptual Modeling*. Springer, 2013. doi:10.1007/978-3-642-39467-6_14.

Peter Royal. What is a business rule? In *The Art of Rules*. Apress, 2023. doi:10.1007/978-1-4842-8992-1_5.

Andres Sacco. MongoDB: Document database. In *Beginning Spring Boot 3*. Apress, 2023. doi:10.1007/978-1-4842-8764-4_8.

Herbert Schildt. Working with java data types. In *Java: A Beginner's Guide, Ninth Edition*. McGraw-Hill Education, 2021. doi:10.1002/9781119696193.ch1.

Matthew F. Tennyson. A study of the data synchronization concern in the observer design pattern. In *2010 2nd International Conference on Software Technology and Engineering*, 2010. doi:10.1109/icste.2010.5608911.

Toby J. Teorey, Sam S. Lightstone, Thomas Nadeau, and H. V. Jagadish. Data model. In *Database Modeling and Design*. Morgan Kaufmann, 2011. doi:10.1017/cbo9780511998225.002.

Balaji Varanasi. Maven lifecycle. In *Introducing Maven*. Apress, 2019. doi:10.1007/978-1-4842-5410-3_5.

Georgina Verity. Relational database management systems and open systems used in the development of Oracle libraries. *Program*, 1993. doi:10.1108/eb047134.

Deepak Vohra. Using a java client with MongoDB. In *Pro Java Clustering and Scalability*. Apress, 2015. doi:10.1007/978-1-4842-1598-2_1.

Indrajit Wijegunaratne and George Fernandez. The three-tier application architecture. In *Distributed Applications Engineering*. Springer, 1998. doi:10.1007/978-1-4471-1550-2_3.

Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. Workload management in database management system: A taxonomy (extended abstract). In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018. doi:10.1109/icde.2018.00269.

Zhuoxing Zhang and Sebastian Link. Synthesizing third normal form schemata that minimize integrity maintenance and update overheads: Parameterizing 3nf by the numbers of minimal keys and functional dependencies. *Proceedings of the ACM on Management of Data*, 2025. doi:10.1145/3725362.