Sales Management and Reporting

Design Document

Lucas Coelho
lmoreiradesouzacoe2@huskers.unl.edu
Luciano Neto
lguedesdecarvalhon2@huskers.unl.edu
University of Nebraska—Lincoln

Spring 2024
Version 6.0

Object-oriented system, written in Java, that supports YRLess's business model implementing a sales subsystem that will be responsible for managing sales data and producing complete reports in a database-backed system.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of the Design Document.	Lucas Coelho & Luciano Neto	2024-02-16
2.0	Added a UML Chart to the Document.	Lucas Coelho & Luciano Neto	2024-03-01
3.0	Added Database Diagram.	Lucas Coelho & Luciano Neto	2024-03-22
4.0	Added Database integration information.	Lucas Coelho & Luciano Neto	2024-03-22
5.0	Added Data Structure information.	Lucas Coelho & Luciano Neto	2024-04-29
6.0	Final Review	Lucas Coelho & Luciano Neto	2024-05-10

Contents

1 Introduction

LSP Holdings, an investment group, is consolidating various cell phone companies and stores into a new larger company called YRLess, developing new systems for marketing, inventory, billing, and more. As part of this initiative, a sales subsystem is being developed to manage sales data and generate sales reports. A sale can be of a Product with a lease and full purchase option, Data Plan, Voice Plan and Service. Each of these independent sales have their own characteristics.

This document serves as a comprehensive guide to designing and implementing the database-backed application within the sales subsystem while following YRLess' business model. It discusses the design and functionality of the sales subsystem, focusing on its role in managing sales data and generating sales reports. It provide both technical and non-technical perspectives on the project, with a detailed understanding of the sales subsystem's design and functionality.

The document is structured to cover the business rules, functional requirements, database design, object-oriented design, and implementation details of the sales subsystem. It also conforms to the guidelines outlined in the IEEE 1016 standard, ensuring consistency in formatting and content presentation.

1.1 Purpose of this Document

The primary goal of this document is to delineate the project's design and offer a thorough insight into the developmental journey of the YRLess Database-Backed System. It details the sequential steps taken during implementation and clarifies the strategies employed for efficient data management and organization to meet the project's objectives.

It's important to note that this project documentation does not extend support for the installation and configuration processes.

1.2 Scope of the Project

This project focuses on data handling and analysis, specifically involving the storage of sales-related information in a database and the generation of a comprehensive report. It does not include the responsibility of data collection, which is designated to another team specifically assigned for this task.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

IEEE 1016 standard:

- It provides guidelines for describing the architecture of software systems, including both the structural and behavioral aspects. It promotes the use of consistent terminology and notation to improve consistency and readability

Object-Oriented

- A way of writing computer programs that model real-world objects and their interactions. Allows for modular, reusable, and maintainable code, as well as easier conceptualization and modeling of complex systems.

1.3.2 Abbreviations & Acronyms

ERD	Entity Relation Diagram.
UML	Unified Modeling Language.
IEEE	Institute of Electrical and Electronics Engineers

2 Overall Design Description

The overall design of the project is structured to follow rules outlined in YRLess's business model by following good coding practices,

- Creating parameters in the classes to behave as requested;
- Functional requirements as to using databases as a source of data for the application;
- Modeling databases and implementing the data readability to a Java environment for report submission and data processing;
- Object-oriented design with good encapsulation, well-stated inheritance, proper abstraction, and great practices of polymorphism.

To fulfill the cited parameters, the project was designed to have a strict inheritance for the Item class and sub-classes. A great separation of classes was also adopted. The system has 14 different classes, each with its methods and representing its own "real life" model. A various amount of linking between classes was also adopted, for code reusability.

3 Detailed Component Description

This section will cover the specifics of our project's architecture, including the structures of our databases and Java classes.

3.1 Database Design

In designing the database schema for our client's business model, we have carefully considered the various entities and relationships involved in their operations. The Entity Relation Diagram (ERD), Figure 1, reflects these considerations, providing a clear visualization of the structure and connections within the database.

There are four core tables in the system.

1. Person Table: Persons are core entities within the system, encompassing customers, salespersons, managers, and possibly other roles. By storing personal details in a dedicated table, we centralize information facilitating efficient data management and retrieval.
2. Store Table: Stores represent physical locations where transactions occur and products are managed. Each store has a unique code for identification and is associated with a manager and an address. Separating store details into its table allows for better organization and management of store-related information.
3. Sale Table: Sales capture transactions made by customers. By creating a dedicated table for sales, we can efficiently track sale details.
4. Item Table: Items represent products offered for sale by the client's business. Separating item details into its table allows for comprehensive management of product information.

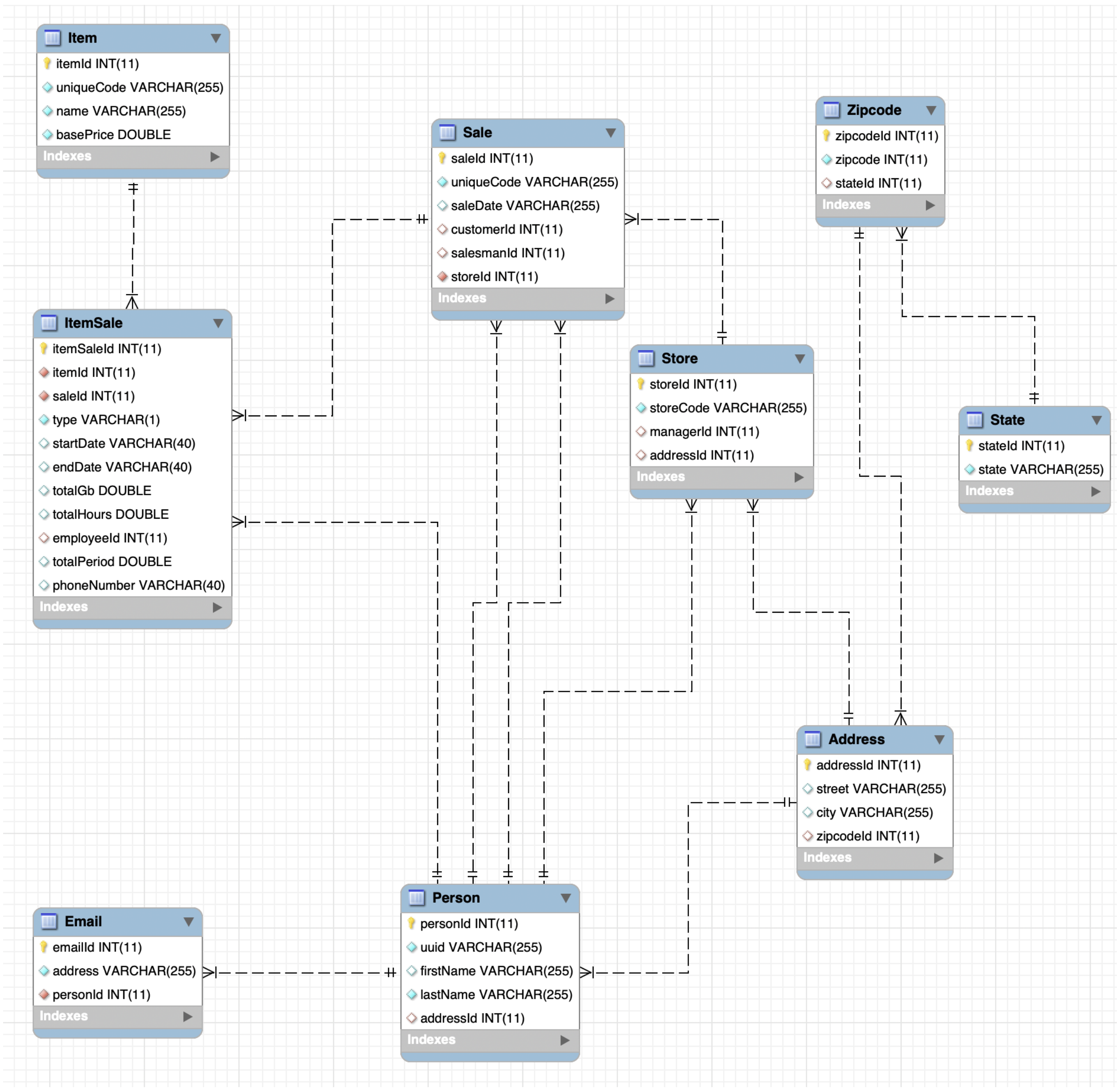

Figure 1: Database ERD Diagram

3.1.1 Component Testing Strategy

In testing the database functionality, we're ensuring its reliability and accuracy across various scenarios. We're covering a range of operations, including basic data retrieval like fetching specific fields from the 'persons' table to more complex tasks such as updating records and detecting potential fraud instances. By testing the retrieval of specific sales records and items, we're verifying the integrity of transaction data.

Additionally, we're assessing the performance of the database by determining the total number of sales at each store and by each employee, along with analyzing the distribution of sales across different product types. This comprehensive testing approach provides valuable insights into the database's functionality, ensuring that it meets both functional and performance requirements.

Furthermore, integration testing with the code is conducted in an external environment provided by the client, ensuring seamless data interaction between the database and the application. This holistic testing strategy ensures the robustness and effectiveness of the database in supporting the project's objectives.

3.2 Class/Entity Model

The project design was thoughtfully planned in an Object-Oriented environment to support a variety of instances, leading to the creation of the classes that can be seen in the diagram below.

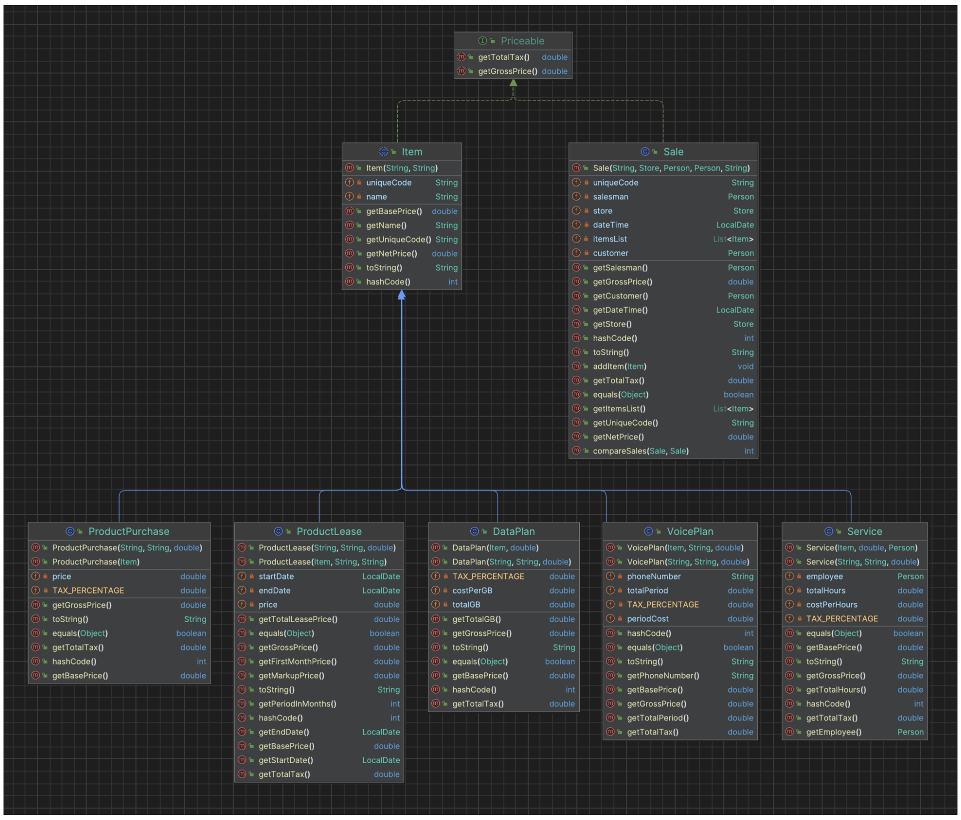

Figure 2: Pricable interface and item modeling strategy.

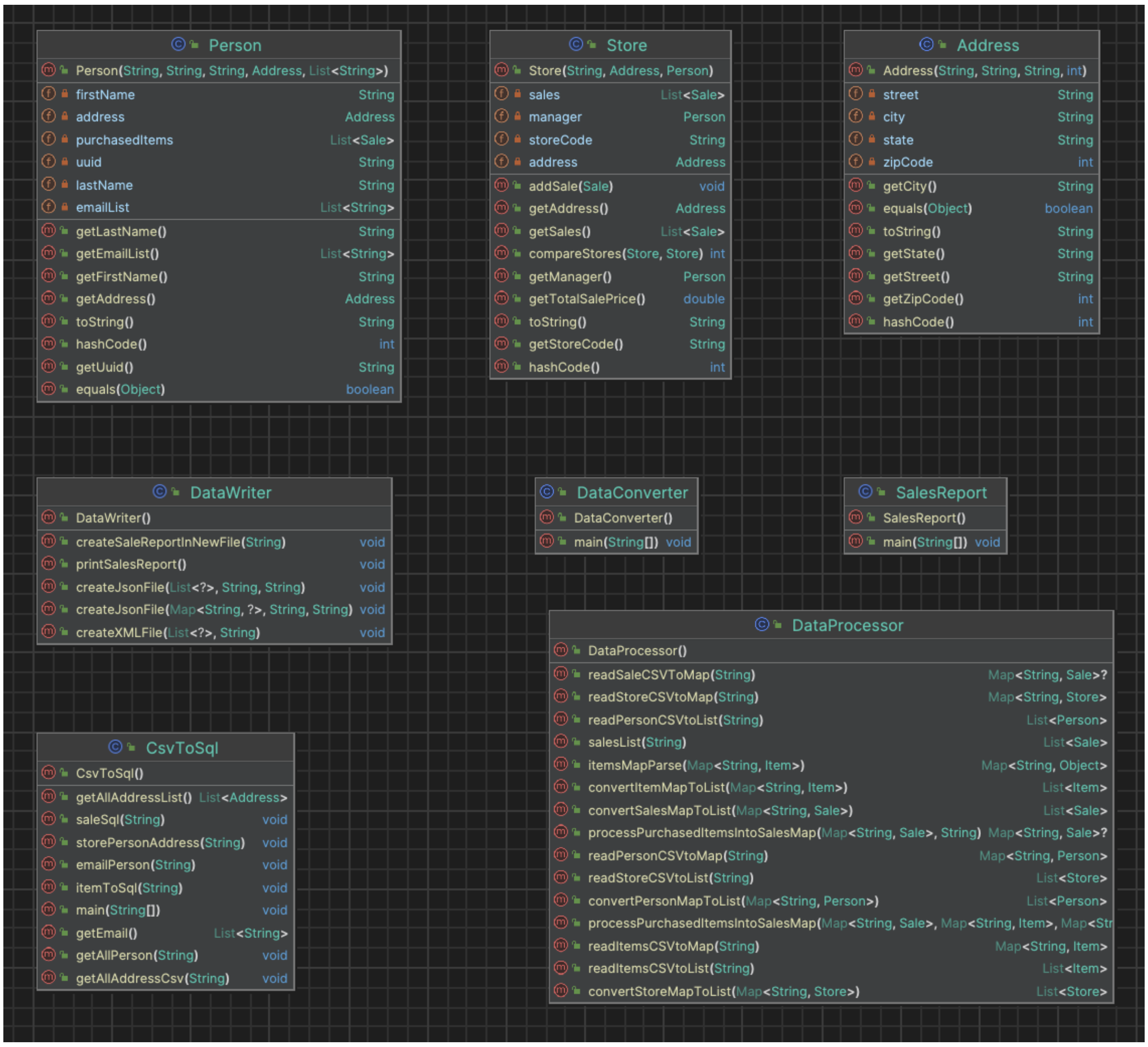
Person
Person(String, String, String, Address, List<String>)
firstName String
address Address
purchasedItems List<Sale>
uuid String
lastName String
emailList List<String>
getLastName() String
getEmailList() List<String>
getFirstName() String
getAddress() Address
toString() String
hashCode() int
getUuid() String
equals(Object) boolean
Store
Store(String, Address, Person)
sales List<Sale>
manager Person
storeCode String
address Address
addSale(Sale) void
getAddress() Address
getSales() List<Sale>
compareStores(Store, Store) int
getManager() Person
getTotalSalePrice() double
toString() String
getStoreCode() String
hashCode() int
Address
Address(String, String, String, int)
street String
city String
state String
zipCode int
getCity() String
equals(Object) boolean
toString() String
getState() String
getStreet() String
getZipCode() int
hashCode() int
DataWriter
DataWriter()
createSaleReportInNewFile(String) void
printSalesReport() void
createJsonFile(List<?>, String, String) void
createJsonFile(Map<String, ?>, String, String) void
createXMLFile(List<?>, String) void
DataConverter
DataConverter()
main(String[]) void
SalesReport
SalesReport()
main(String[]) void
DataProcessor
DataProcessor()
readSaleCSVToMap(String) Map<String, Sale>?
readStoreCSVtoMap(String) Map<String, Store>
readPersonCSVtoList(String) List<Person>
salesList(String) List<Sale>
itemsMapParse(Map<String, Item>) Map<String, Object>
convertItemMapToList(Map<String, Item>) List<Item>
convertSalesMapToList(Map<String, Sale>) List<Sale>
processPurchasedItemsIntoSalesMap(Map<String, Sale>, String) Map<String, Sale>?
readPersonCSVtoMap(String) Map<String, Person>
readStoreCSVtoList(String) List<Store>
convertPersonMapToList(Map<String, Person>) List<Person>
processPurchasedItemsIntoSalesMap(Map<String, Sale>, Map<String, Item>, Map<Str
readItemsCSVtoMap(String) Map<String, Item>
readItemsCSVtoList(String) List<Item>
convertStoreMapToList(Map<String, Store>) List<Store>
CsvToSql
CsvToSql()
getAllAddressList() List<Address>
saleSql(String) void
storePersonAddress(String) void
emailPerson(String) void
itemToSql(String) void
main(String[]) void
getEmail() List<String>
getAllPerson(String) void
getAllAddressCsv(String) void

Figure 3: Other classes with functionalities and relations

The implementation of an interface priceable guarantees a great polymorphism if necessary. Other classes don't need to be tied in a hierarchy for better flexibility of the code, so it was left as an independent class.

3.2.1 Component Testing Strategy

The Testing Strategy will consist of several important factors:

- Utilization of a Unit Testing approach.
- Employing JUnit as the framework for developing and evaluating the assertiveness of test cases.

- Designing six test cases for each method, covering various scenarios including large and correct data samples, incomplete data samples of different sizes, and invalid data samples.
- Ensuring that any output report aligns with the expected results, with code revisions made if necessary.
- Conducting tests specifically on methods with dependencies, ensuring comprehensive coverage.
- Testing the time taken for database queries, setting time limits for actions, and exploring edge cases to assess the program's performance.
- Implementing stress testing by increasing data volume beyond typical scenarios to evaluate system response under extreme conditions.
- Performing security testing to identify and address vulnerabilities, safeguarding sensitive information.

The goal of this testing is to evaluate the accuracy of the classes and methods within the project and to generate a report on its performance and assertiveness.

3.3 Database Interface

The API is the intermediary between users and our system, (using JDBC) helping the data exchange. It begins by validating incoming data, ensuring its integrity by checking formats and lengths. Once validated, the API routes data to the appropriate processing function, executing necessary business logic to fulfill user requests, which may include querying databases or calling external services.

Throughout this process, error handling mechanisms maintain system stability, logging errors and providing informative messages to users to ensure a better experience. After processing, the API transforms data into the required format, constructs the response object, and delivers it to report all the information for the sales system. Continuous logging of relevant information enables real-time monitoring.

3.3.1 Component Testing Strategy

Test Cases:

- **Large Data**: Two cases with correct, complete data.
- **Incomplete Data**: One case with incomplete data.
- **Small Data**: One case with minimal data.
- **Invalid Data**: Two cases with invalid data.

Specific Scenarios:

- **Query Time**: Test database query execution time.
- **Data Consistency**: Ensure retrieved data matches expected values.

Expected Outcomes:

- **Data Accuracy**: Ensure data matches expectations.
- **Performance Validation**: Confirm system meets performance requirements based on time of execution.

3.4 Design & Integration of a Sorted List Data Structure

The list implementation is node-based. It uses a private static class `Node<T>` to represent each element in the list. Each node contains a reference to the next and previous nodes.

The interface includes methods for adding, removing, and accessing elements in the list (`add`, `remove`, `get`). The list is sorted based on the provided comparator or natural ordering if no comparator is provided. The `add` method ensures that elements are inserted into the list in their sorted order, and the method `remove` ensures to set the right ordering also, therefore, it always **maintains** its ordering.

It uses general class parameters `<T>` to allow the list to store elements of any type. This makes the implementation more flexible and reusable since it can be used to store different types of objects without the need for separate implementations. It follows the following generality concepts.

1. **Generic and User-friendly:**
    - The implementation aims to be generic, meaning it should work with any type of object that you want to store in the list. This allows users to utilize the `SortedLinkedList` with various types of objects without needing to create separate implementations for each type.
    - Additionally, it aims to be user-friendly, meaning it should be easy for developers to use and understand. By providing a single implementation that handles sorting internally.
2. **Comparator Usage:**
    - The `SortedLinkedList` class allows users to provide a comparator object during instantiation. This comparator defines the sorting order of the elements in the list.
    - However, providing a comparator is optional in certain cases. Specifically, when the objects being stored in the list implement the `Comparable` interface and have a `compareTo` method defined, the comparator is not necessary.

3. **Handling Cases Without Comparable:**
    - If the objects being stored in the list do not implement `Comparable` or do not have a `compareTo` method, the comparator becomes necessary. This is because the list needs some way to determine the sorting order of these objects.
    - In such cases, users must provide a comparator when creating the list. This comparator defines the custom sorting logic that the list will use to order the elements. If a comparator is provided, it will override all-natural ordering, sorting by the comparator instead.

To be even more user-friendly, the iterator interface is implemented, providing a way to access elements of the list one by one, abstracting away the details of the list's implementation. The iterator enhances the usability of `SortedLinkedList` by providing a clean and efficient way to iterate over its elements.

Overall, the `SortedLinkedList` class offers a flexible, user-friendly, and efficient solution for managing sorted lists in Java.

3.4.1 Component Testing Strategy

Unit Tests: Test adding elements, removing elements, and getting elements with different test sizes.

Comparator Tests: Test custom comparator to ensure that the list sorts elements correctly using a custom comparator and validate that the list sorts elements correctly based on their natural ordering if no comparator is given.

Iterator Tests: Test iterating over elements to verify that the iterator iterates over all elements in the list correctly. Also ensure that the iterator throws a `NoSuchElementException` when there are no more elements.

Edge Cases and Boundary Tests: Test adding and removing from an empty List to verify behavior when adding and removing elements from an empty list. Test adding and removing at boundaries to check behavior when adding and removing elements at the beginning and end of the list. Test adding and removing duplicate elements to ensure correct handling of duplicate elements in the list.

Performance Tests: Test large data sets to evaluate the performance of the list with a large number of elements.

These tests cover a range of scenarios to ensure the correctness, robustness, and performance of the 'SortedLinkedList' implementation.

4 Changes & Refactoring

Initially, we developed a program for processing plain text, primarily in CSV format, parsing the data into JSON and XML formats. Subsequently, we utilized this capability to generate sales reports within our system by manipulating CSV data and establishing relationships between files. Throughout the process, we iteratively refined our class structures, accommodating changing requirements by adding or removing components to improve usability and efficiency.

As our system evolved, we transitioned from file-based storage to a more robust and reliable database solution. This shift enabled us to implement new functionalities for querying and managing the data, while also establishing a connection factory to enhance database connectivity. To optimize performance, we implemented static functions to cache previously queried data, reducing the need for repetitive queries and significantly improving efficiency in data retrieval operations.

References

[1] IEEE standard for information technology–systems design–software design descriptions. *IEEE STD 1016-2009*, pages 1–35, July 2009.