DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

Invoice Reporting System

FarMarT

Keyik Annagulyyeva and Yusup Orazov

May 17th, 2023

This design document outlines the development of a Invoice Reporting System for FarMarT, focusing on object-oriented design, database design, and implementation of classes and database schemas. The project involves creating Java classes to model entities, reading data from CSV files, creating object instances, and exporting data in XML and/or JSON format, as well as designing and implementing a MySQL database to support the application.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Keyik Annagulyyeva Yusup Orazov	2023/17/02
2.0	Add functionality, integrate all the classes and produce three types of reports: summary of all sales, summary for each store, and details of each individual invoice.	Keyik Annagulyyeva Yusup Orazov	2023/03/03
3.0	Updated system architecture and database structure for clarity. Improved Java class descriptions and calculations.	Keyik Annagulyyeva Yusup Orazov	2023/03/24
4.0	Reviewed and enhanced system architecture and database structure. Refined Java class descriptions, calculations, and reports.	Keyik Annagulyyeva Yusup Orazov	2023/04/06

Contents

1. Introduction

This design document presents the development plan for a Invoice Reporting System for FarMarT, a regional B2B chain of stores supplying farm equipment and services. The project was initiated by Pablo Myers, who recently took over the family business and identified the need for technological upgrades in various aspects of its operations. The Java-based, object-oriented system aims to streamline sales processes, improve efficiency, and cater to FarMarT's unique business requirements, benefiting the customers and the company alike.

The project is divided into multiple phases, focusing on data representation, electronic data interchange, summary report generation, database design and connectivity, persistence, and implementation of a sorted list ADT. The system will accurately manage and organize entities like customers, salespersons, products, services, and invoices while considering necessary calculations and tax rules for generating invoices.

The document provides an overview of the project, objectives, and the approach for each development phase, with updates reflecting new information. It aims to provide a comprehensive understanding of the project's background, and the technical aspects involved in developing an efficient Invoice Reporting System for FarMarT.

1. Purpose of this Document

This document outlines the system's design and development process, including the database schema design, component testing strategies, and the integration of data structures into the system. Additionally, it details the testing approaches employed throughout the development process and any changes or refactoring made to the system.

1.2.Scope of the Project

The Invoice Reporting System is designed to provide FarMarT with a comprehensive solution that supports its business model. The system accurately represents and organizes sales data, facilitating efficient storage, management, and retrieval. The system's features include data representation, electronic data interchange, data summary report generation, database design, database connectivity, and API implementation. The system's scope includes the creation of a database schema, database connectivity, and the integration of a custom data structure. The project's scope does not include any external integrations with other systems or any other features that are not related to Invoice Reporting System.

1.3.Definitions, Acronyms, Abbreviations

1.3.1. Definitions

Entity - A distinct object or concept in the problem domain.

Serialization - The process of converting an object's state to a byte stream.

1.3.2. Abbreviations & Acronyms

CSV (Comma-Separated Values): A simple file format used to store tabular data, such as a spreadsheet or database, where each data field is separated by a comma.

JSON (JavaScript Object Notation): A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate, often used in web applications for data transmission. [2]

XML (Extensible Markup Language): A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable, used to store and transport data. [1]

OOP (Object-Oriented Programming): A programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (also known as attributes), and code, in the form of procedures (also known as methods).

EDI (Electronic Data Interchange): A system for exchanging business documents between companies using a standardized electronic format, replacing traditional paper-based exchanges.

ADT (Abstract Data Type): A high-level description of a collection of data and the operations that can be performed on that data, without specifying how the data will be organized or the algorithms that will be used for the operations.

2. Overall Design Description

The project comprised two main components: Java classes representing various entities and a MySQL database for data storage and management. These Java classes conformed to OOP principles and utilized serialization libraries, whereas the database was designed with a focus on data integrity and normalization. The system was adapted to load data from the MySQL database instead of flat files, employing JDBC for seamless database connectivity. The design aimed to accurately represent and organize sales data, facilitating efficient storage, management, and retrieval of information to support FarMarT's business model.

1. Alternative Design Options

An alternative design option considered involved using a NoSQL database instead of a relational database. Although NoSQL databases offer flexibility and scalability, a MySQL database was chosen due to the structured nature of the sales data, which is better suited for a relational database.

3. Detailed Component Description

This section provides a comprehensive breakdown of the primary components of the Invoice Reporting System, elaborating on their functionality, design, and interactions. This detailed overview enables a thorough understanding of the system's structure and facilitates its successful implementation.

This project can be broken down into five main phases:

Data Representation and Electronic Data Interchange (EDI): In this phase, Java classes are created to model entities like customers, salespersons, products, services, and invoices. These classes are used to parse data from CSV files, creating object instances from this data. The objects are then serialized into XML or JSON formats, facilitating data exchange between different subsystems.

Summary Report: This phase refines the object models and establishes relationships between them, allowing for the creation of a summary report that aggregates sales data. This report serves to provide an overview of the company's sales performance.

Database Design: Here, a relational database is designed to model the objects and support the application. This design allows for efficient storage, management, and retrieval of sales data, ensuring data integrity and consistency.

Database Connectivity: In this phase, the code is updated to load objects into the database rather than from flat files. This ensures that the sales data is always up-to-date and synchronized, which leads to more accurate and reliable reporting.

Database Persistence and Sorted List ADT: The final phase involves implementing an API for data persistence in the database, streamlining the process of storing and retrieving data. This phase also involves the creation of a Sorted List Abstract Data Type (ADT) to organize and manage sales data efficiently.

Throughout these phases, data is taken from CSV files, turned into objects, serialized into XML or JSON, loaded into a database, and managed through an API and a Sorted List ADT. This provides a comprehensive, database-backed application for managing the sales of FarMarT.

1. Database Design

The database schema consisted of tables representing Services, Products, Invoice Items, Addresses, Stores, Invoices, Equipment, and Persons. These tables utilized primary, non-primary, and foreign keys to ensure data integrity, with relationships designed to support many-to-one and many-to-many connections. The schema also conformed to data normalization principles:

- Person table: This table represented people involved in the system, such as customers, salespersons, and store managers. It maintained a one-to-many relationship with the Email table (a person can have multiple email addresses) and a one-to-one relationship with the Address table (a person has one address).

- Email table: This table stored email addresses of people in the system. It maintained a many-to-one relationship with the Person table (an email address belongs to a single person) and facilitated communication with customers and salespersons.

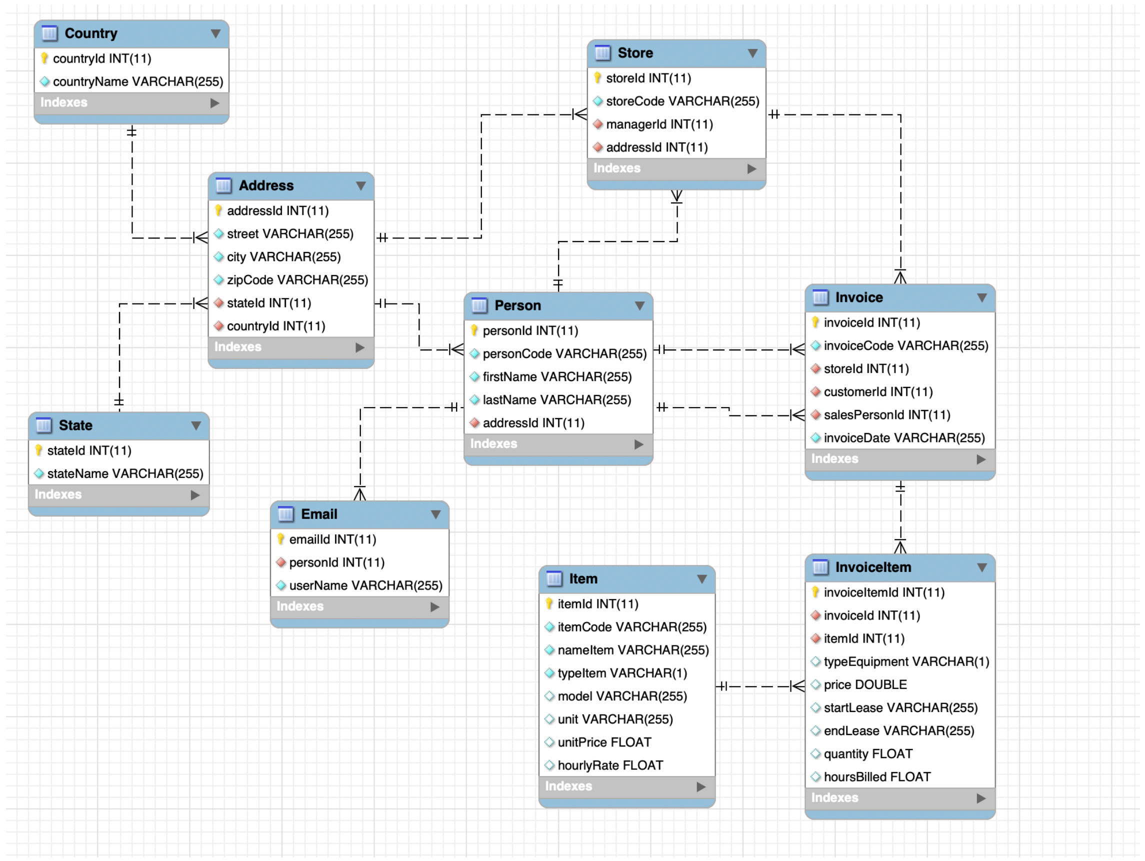
- Address table: This table stored addresses for people and stores. It maintained a one-to-many relationship with the Person and Store tables (an address can be associated with multiple people or stores) and a many-to-one relationship with the State and Country tables (an address belongs to a single state and country).

- State and Country tables: These tables stored information about states and countries, respectively. They maintained a one-to-many relationship with the Address table (a state or country can have multiple addresses) and provided location information for addresses.

- Store table: This table represented the stores in the system. It maintained a one-to-one relationship with the Address table (a store has one address) and a one-to-many relationship with the Invoice table (a store can have multiple invoices). It also maintained a one-to-one relationship with the Person table, as each store has a manager.

- Item table: This table stored information about items or services offered by FarMarT. It maintained a many-to-many relationship with the Invoice table through the InvoiceItem table (an item can be part of multiple invoices, and an invoice can contain multiple items).

- InvoiceItem table: This table acted as a junction table between the Invoice and Item tables, representing the items in an invoice. It had a many-to-one relationship with both the Invoice and Item tables (an InvoiceItem record belongs to a single invoice and a single item), and supported the storage of



ER diagram

information about item quantities, lease start and end dates, and hours billed for each item in an invoice.

1.1. Component Testing Strategy

The testing process involved creating sample data and inserting it into the database tables. This helped to verify that the database schema design and implementation were correct and that the database functioned as intended. Additionally, testing included verifying the database's ability to handle and store data, as well as its ability to retrieve and display data accurately. This was accomplished by querying the database and comparing the results with the expected output. The outcomes of the tests were analyzed and any issues were addressed through refactoring and redesigning of the database schema and implementation. Testing was an ongoing process throughout the development cycle to ensure that the database continued to function correctly as the application evolved.

2. Class/Entity Model

The Invoice Reporting System for FarMarT aims to provide a comprehensive and database-backed application to support their business model. This section focuses on the design of various Java classes/ entities and their relationships within the system:

The Person class represents individuals within the Invoice Reporting System. It contains attributes such as personId, personCode, firstName, lastName, and emails. The Person class serves as a data container to store personal information of customers and salespersons. It includes methods to access and retrieve these attributes. The association between the Person class and other classes, such as Invoice and Store, is defined through references.

The Store class represents a store within the Invoice Reporting System. It contains attributes such as storeId, storeCode, managerId (Person), address, and a collection of associated invoices. The Store class provides methods to access and retrieve these attributes, as well as calculate revenue, sales summaries, and other relevant information. The Store class is responsible for managing store-specific data and operations, such as adding invoices and generating sales reports..

The Item class is an abstract class that serves as a base for different types of sale items in the system, including products, services, and equipment. The Item class defines common attributes such as code, name, and type. It also defines abstract methods for calculating tax, incomplete price, and final total. The purpose of using inheritance is to provide a common interface for handling different types of items and to facilitate code reuse. The Product and Service classes extend the Item class and provide specific implementations for their respective item types. The Equipment class, mentioned below, is a specialization of Item and introduces additional attributes and behavior specific to equipment items.

The Invoice class represents an invoice in the Invoice Reporting System. It contains attributes such as invoiceId, invoiceCode, itemCode, invoiceDate, and references to related entities such as Store, Customer (Person), Salesperson (Person), and InvoiceItems. The Invoice class also provides methods to retrieve and manipulate these attributes, as well as calculate various values like incomplete price, tax, and final total. Additionally, it includes methods to generate sales summary reports and handle invoice-related operations.

The ConnectionFactory class handles the database connection for the Invoice Reporting System. It provides a static method, getConnection(), to retrieve the connection instance. The connection details, such as URL, username, and password, are defined as private static attributes within the class.

The DatabaseLoader class is responsible for loading data from the database for various entities in the system, including Persons, Stores, Invoices, Items, and InvoiceItems. It contains static methods to establish connections to the database and retrieve data for each entity. The loading process involves executing appropriate queries and mapping the retrieved data to corresponding Java objects.

The Equipment class represents equipment items. It extends the Item class and introduces additional attributes such as model. The class provides methods to access these attributes and implements the abstract methods defined in the Item class. The Equipment class is designed to handle the two types of equipment items, namely Purchase and Lease. For equipment items, the total is determined based on whether it is a purchase or a lease. If it is a purchase, the purchase price of the equipment is used as the total. If it is a lease, additional calculations are performed considering the 30-day fee, start date, and end date of the lease. The total is derived by multiplying the 30-day fee by the number of days in the lease period. Taxes are calculated based on the item type and associated rules. The specific rules for calculating totals and taxes differ for each type, and these rules are implemented within the Equipment class.

InvoiceReport: The InvoiceReport.java class is the driver class that generates various summary reports based on the data stored in the other classes. It generates summary reports in three different methods: ‘summaryReportByTotal()’, ‘storeSummary()’, and ‘invoiceSummary()’.

The SortedList class represents a sorted list data structure. It is a generic class that can hold objects of any type. The class includes attributes such as info (an array to store the elements), size (the number of elements in the list), and a comparator to define the sorting order. The SortedList class provides methods to add and remove elements, retrieve the size, and iterate over the list.

The InvoiceComparator class contains static methods to define different comparators for sorting invoices. It includes methods such as customerNameComparator, totalValueComparator, and storeAndSalespersonComparator. These comparators are used to sort invoices based on different criteria, such as customer name, total value, or store and salesperson combination.

The InvoiceData class provides static methods to interact with the database and perform various operations related to data manipulation. It includes methods to add persons, stores, products, services, and invoices to the database. Additionally, it provides methods to retrieve specific data from the database, such as person IDs, store IDs, and item IDs. The InvoiceData class ensures data consistency and integrity by handling database transactions and operations.

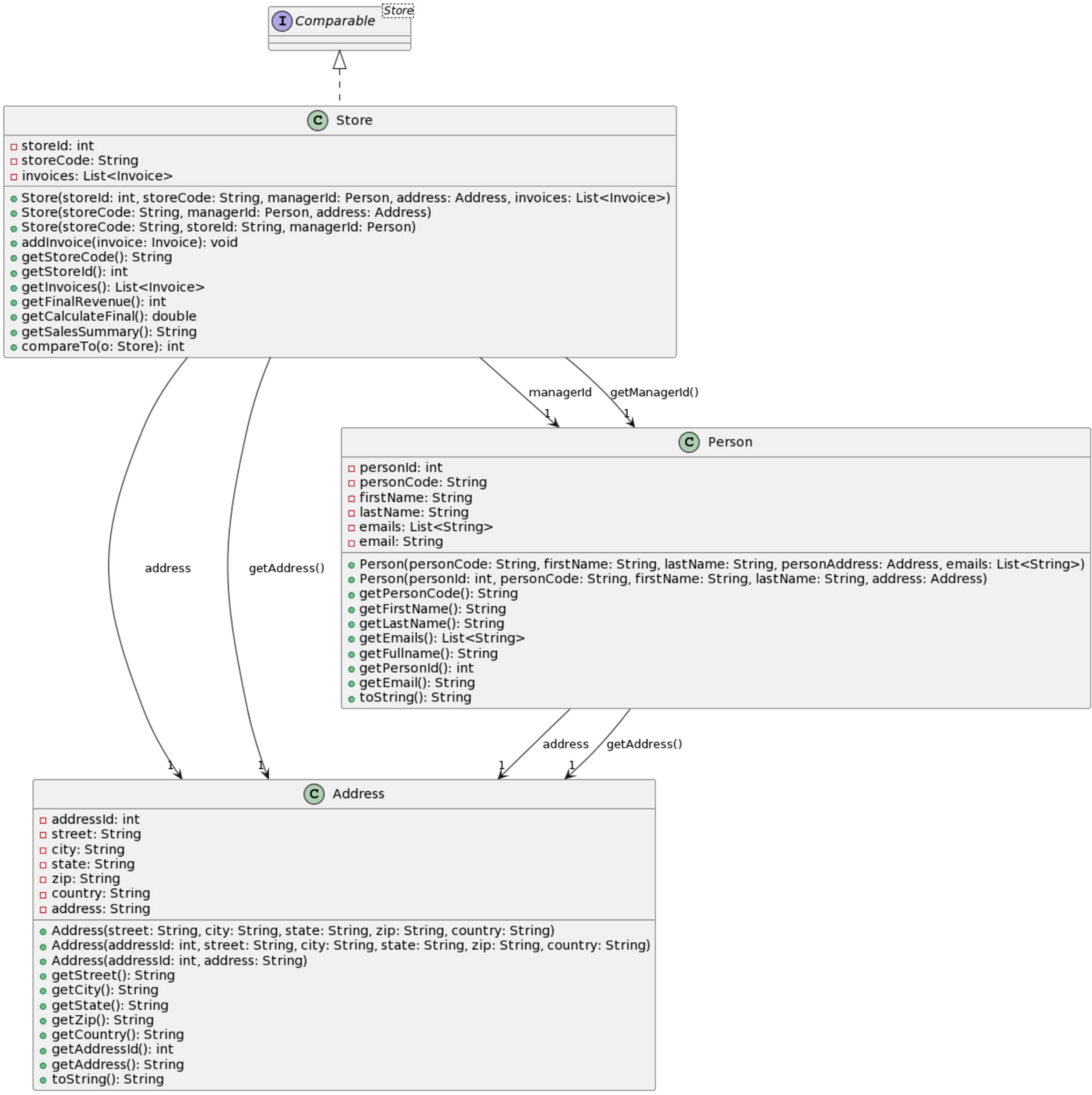
The sequence diagram in the UML documentation illustrates the interactions between the various classes in the system when generating a summary report. As follows:

The InvoiceReport class initiates the data loading process by invoking the LoaderFiles class, which populates the relevant classes, including Persons, Stores, Items, Invoices, and InvoiceItems. Once the data is successfully loaded, the InvoiceReport class proceeds to generate the desired summary report by calling the corresponding method, such as summaryReportByTotal, storeSummary, or invoiceSummary.

During the report generation process, the InvoiceReport class retrieves and processes the required data from the associated classes, including Person, Store, Item, Invoice, and InvoiceItem, in order to compute the necessary totals and summaries. Finally, the InvoiceReport class formats and presents the report, showcasing the calculated results in an organised and easily comprehensible manner.

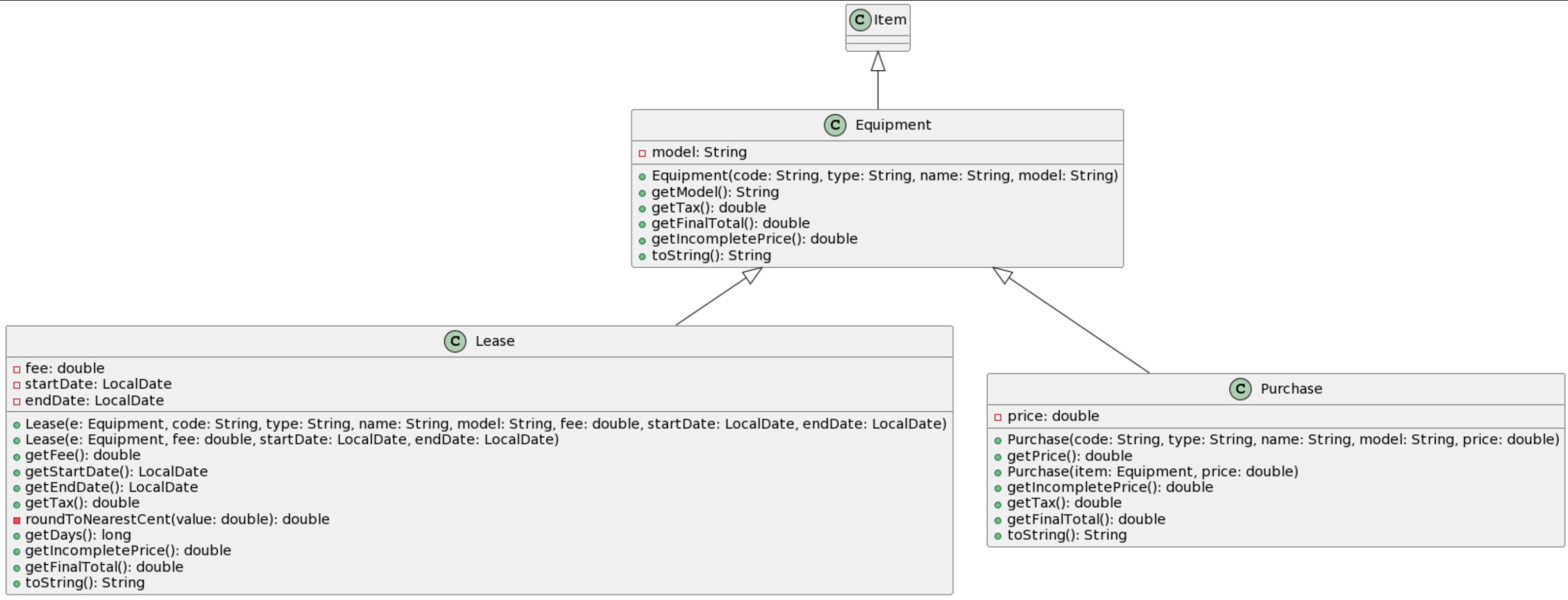
The design choices made in the system, such as using inheritance for item types and defining associations between entities, provide several benefits. Inheritance allows for code reuse and provides a common interface for handling different types of items. Associations and relationships between classes enable data organization and retrieval, allowing for efficient and meaningful operations within the system

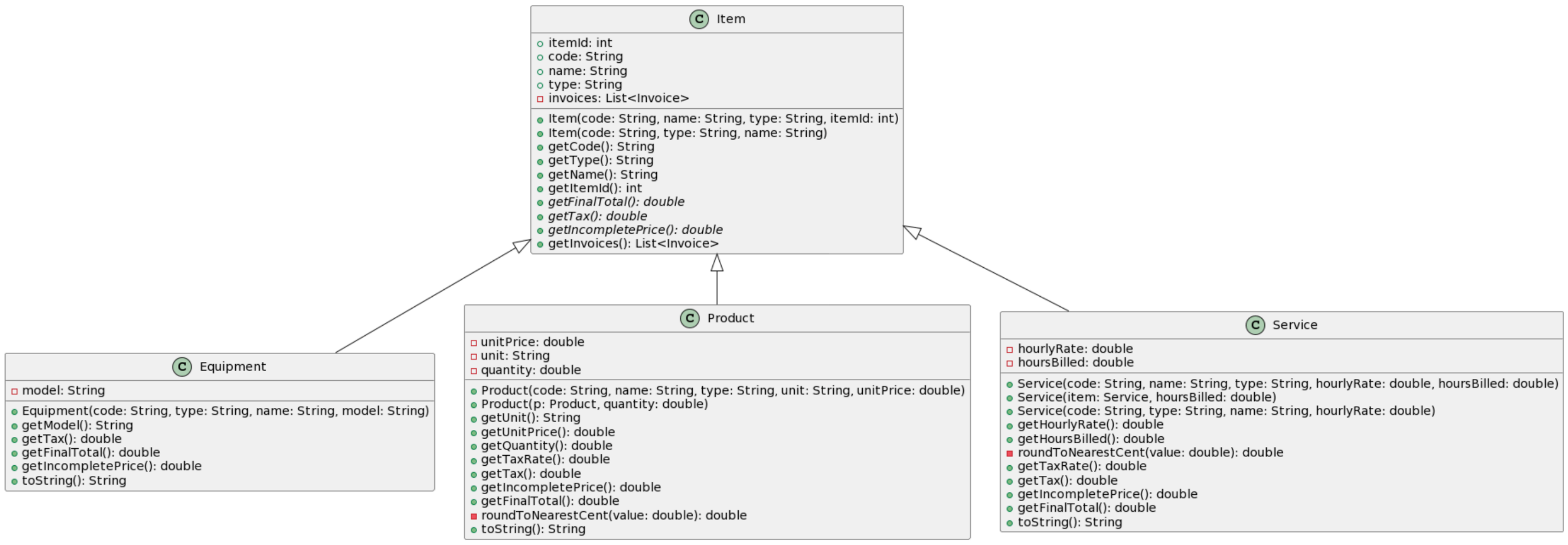
The program is divided into different sections based on the responsibilities and functionalities of each class. The Address, ConnectionFactory, and DatabaseLoader classes handle database-related operations and data retrieval. The Equipment, Product, and Service classes represent different types of items and provide specific implementations for calculating totals and taxes. The Person and Store classes hold information about individuals and stores, respectively, and manage associated data and operations. The Invoice class is responsible for representing and manipulating invoice data, as well as generating reports. The SortedList and InvoiceComparator classes provide data structures and sorting mechanisms. The InvoiceData class acts as a mediator for database interactions and data manipulation.

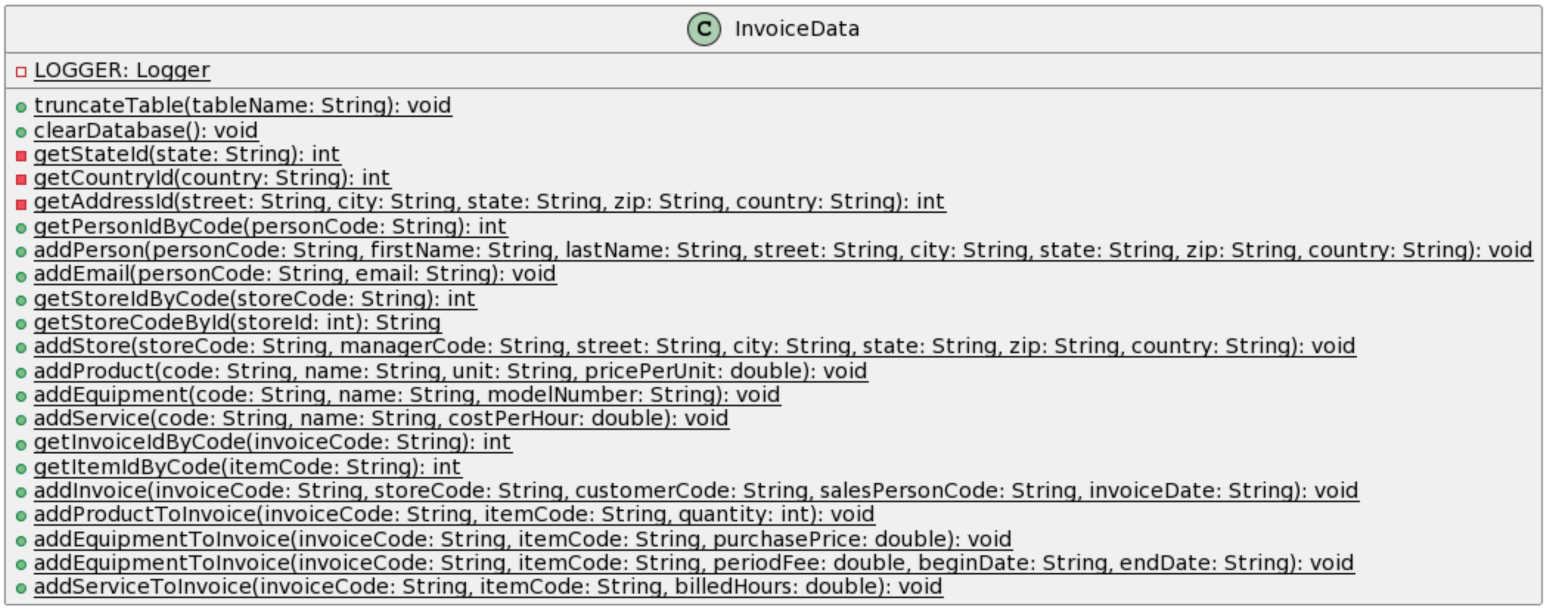


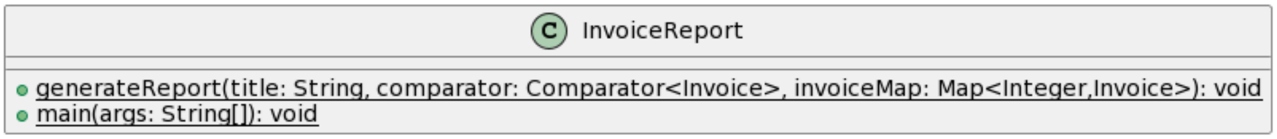
Comparable
Store
Store
- storeId: int
- storeCode: String
- invoices: List<Invoice>
+ Store(storeId: int, storeCode: String, managerId: Person, address: Address, invoices: List<Invoice>)
+ Store(storeCode: String, managerId: Person, address: Address)
+ Store(storeCode: String, storeId: String, managerId: Person)
+ addInvoice(invoice: Invoice): void
+ getStoreCode(): String
+ getStoreId(): int
+ getInvoices(): List<Invoice>
+ getFinalRevenue(): int
+ getCalculateFinal(): double
+ getSalesSummary(): String
+ compareTo(o: Store): int
managerId
1
getManagerId()
1
address
1
getAddress()
1
Person
- personId: int
- personCode: String
- firstName: String
- lastName: String
- emails: List<String>
- email: String
+ Person(personCode: String, firstName: String, lastName: String, personAddress: Address, emails: List<String>)
+ Person(personId: int, personCode: String, firstName: String, lastName: String, address: Address)
+ getPersonCode(): String
+ getFirstName(): String
+ getLastName(): String
+ getEmails(): List<String>
+ getFullname(): String
+ getPersonId(): int
+ getEmail(): String
+ toString(): String
address
1
getAddress()
1
Address
- addressId: int
- street: String
- city: String
- state: String
- zip: String
- country: String
- address: String
+ Address(street: String, city: String, state: String, zip: String, country: String)
+ Address(addressId: int, street: String, city: String, state: String, zip: String, country: String)
+ Address(addressId: int, address: String)
+ getStreet(): String
+ getCity(): String
+ getState(): String
+ getZip(): String
+ getCountry(): String
+ getAddressId(): int
+ getAddress(): String
+ toString(): String

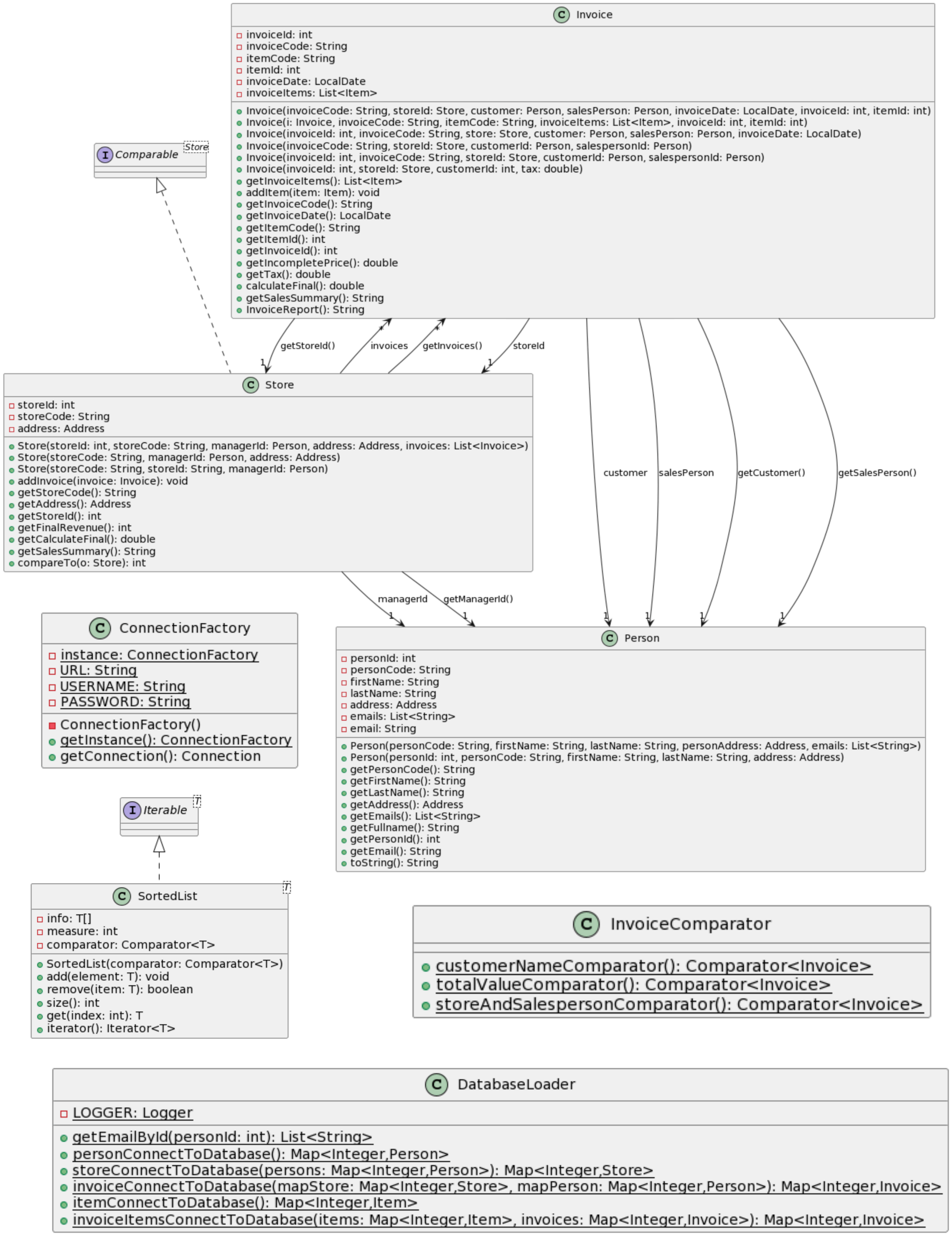
Overall, the Classes and Entities section highlights the interconnected nature of the system components and their significance in providing a comprehensive Invoice Reporting solution for FarMarT. The system enables the company to effectively track sales data, analyze performance, make informed decisions, and ultimately, thrive in their market


Item
Equipment
- model: String
+ Equipment(code: String, type: String, name: String, model: String)
+ getModel(): String
+ getTax(): double
+ getFinalTotal(): double
+ getIncompletePrice(): double
+ toString(): String
Lease
- fee: double
- startDate: LocalDate
- endDate: LocalDate
+ Lease(e: Equipment, code: String, type: String, name: String, model: String, fee: double, startDate: LocalDate, endDate: LocalDate)
+ Lease(e: Equipment, fee: double, startDate: LocalDate, endDate: LocalDate)
+ getFee(): double
+ getStartDate(): LocalDate
+ getEndDate(): LocalDate
+ getTax(): double
- roundToNearestCent(value: double): double
+ getDays(): long
+ getIncompletePrice(): double
+ getFinalTotal(): double
+ toString(): String
Purchase
- price: double
+ Purchase(code: String, type: String, name: String, model: String, price: double)
+ getPrice(): double
+ Purchase(item: Equipment, price: double)
+ getIncompletePrice(): double
+ getTax(): double
+ getFinalTotal(): double
+ toString(): String


Item
itemId: int
code: String
name: String
type: String
- invoices: List<Invoice>
+ Item(code: String, name: String, type: String, itemId: int)
+ Item(code: String, type: String, name: String)
+ getCode(): String
+ getType(): String
+ getName(): String
+ getItemId(): int
+ getFinalTotal(): double
+ getTax(): double
+ getIncompletePrice(): double
+ getInvoices(): List<Invoice>
Equipment
- model: String
+ Equipment(code: String, type: String, name: String, model: String)
+ getModel(): String
+ getTax(): double
+ getFinalTotal(): double
+ getIncompletePrice(): double
+ toString(): String
Product
- unitPrice: double
- unit: String
- quantity: double
+ Product(code: String, name: String, type: String, unit: String, unitPrice: double)
+ Product(p: Product, quantity: double)
+ getUnit(): String
+ getUnitPrice(): double
+ getQuantity(): double
+ getTaxRate(): double
+ getTax(): double
+ getIncompletePrice(): double
+ getFinalTotal(): double
- roundToNearestCent(value: double): double
+ toString(): String
Service
- hourlyRate: double
- hoursBilled: double
+ Service(code: String, name: String, type: String, hourlyRate: double, hoursBilled: double)
+ Service(item: Service, hoursBilled: double)
+ Service(code: String, type: String, name: String, hourlyRate: double)
+ getHourlyRate(): double
+ getHoursBilled(): double
- roundToNearestCent(value: double): double
+ getTaxRate(): double
+ getTax(): double
+ getIncompletePrice(): double
+ getFinalTotal(): double
+ toString(): String


InvoiceData
- LOGGER: Logger
+ truncateTable(tableName: String): void
+ clearDatabase(): void
- getStateId(state: String): int
- getCountryId(country: String): int
- getAddressId(street: String, city: String, state: String, zip: String, country: String): int
+ getPersonIdByCode(personCode: String): int
+ addPerson(personCode: String, firstName: String, lastName: String, street: String, city: String, state: String, zip: String, country: String): void
+ addEmail(personCode: String, email: String): void
+ getStoreIdByCode(storeCode: String): int
+ getStoreCodeById(storeId: int): String
+ addStore(storeCode: String, managerCode: String, street: String, city: String, state: String, zip: String, country: String): void
+ addProduct(code: String, name: String, unit: String, pricePerUnit: double): void
+ addEquipment(code: String, name: String, modelNumber: String): void
+ addService(code: String, name: String, costPerHour: double): void
+ getInvoiceIdByCode(invoiceCode: String): int
+ getItemIdByCode(itemCode: String): int
+ addInvoice(invoiceCode: String, storeCode: String, customerCode: String, salesPersonCode: String, invoiceDate: String): void
+ addProductToInvoice(invoiceCode: String, itemCode: String, quantity: int): void
+ addEquipmentToInvoice(invoiceCode: String, itemCode: String, purchasePrice: double): void
+ addEquipmentToInvoice(invoiceCode: String, itemCode: String, periodFee: double, beginDate: String, endDate: String): void
+ addServiceToInvoice(invoiceCode: String, itemCode: String, billedHours: double): void


InvoiceReport
+ generateReport(title: String, comparator: Comparator<Invoice>, invoiceMap: Map<Integer,Invoice>): void
+ main(args: String[]): void


Invoice
invoiceId: int
invoiceCode: String
itemCode: String
itemId: int
invoiceDate: LocalDate
invoiceItems: List<Item>
Invoice(invoiceCode: String, storeId: Store, customer: Person, salesPerson: Person, invoiceDate: LocalDate, invoiceId: int, itemId: int)
Invoice(i: Invoice, invoiceCode: String, itemCode: String, invoiceItems: List<Item>, invoiceId: int, itemId: int)
Invoice(invoiceId: int, invoiceCode: String, store: Store, customer: Person, salesPerson: Person, invoiceDate: LocalDate)
Invoice(invoiceCode: String, storeId: Store, customerId: Person, salespersonId: Person)
Invoice(invoiceId: int, invoiceCode: String, storeId: Store, customerId: Person, salespersonId: Person)
Invoice(invoiceId: int, storeId: Store, customerId: int, tax: double)
getInvoiceItems(): List<Item>
addItem(item: Item): void
getInvoiceCode(): String
getInvoiceDate(): LocalDate
getItemCode(): String
getItemId(): int
getInvoiceId(): int
getIncompletePrice(): double
getTax(): double
calculateFinal(): double
getSalesSummary(): String
InvoiceReport(): String
Comparable
Store
getStoreId()
1
*
invoices
*
getInvoices()
storeId
1
customer
salesPerson
getCustomer()
getSalesPerson()
Store
storeId: int
storeCode: String
address: Address
Store(storeId: int, storeCode: String, managerId: Person, address: Address, invoices: List<Invoice>)
Store(storeCode: String, managerId: Person, address: Address)
Store(storeCode: String, storeId: String, managerId: Person)
addInvoice(invoice: Invoice): void
getStoreCode(): String
getAddress(): Address
getStoreId(): int
getFinalRevenue(): int
getCalculateFinal(): double
getSalesSummary(): String
compareTo(o: Store): int
managerId
1
getManagerId()
1
1
1
1
1
ConnectionFactory
instance: ConnectionFactory
URL: String
USERNAME: String
PASSWORD: String
ConnectionFactory()
getInstance(): ConnectionFactory
getConnection(): Connection
Person
personId: int
personCode: String
firstName: String
lastName: String
address: Address
emails: List<String>
email: String
Person(personCode: String, firstName: String, lastName: String, personAddress: Address, emails: List<String>)
Person(personId: int, personCode: String, firstName: String, lastName: String, address: Address)
getPersonCode(): String
getFirstName(): String
getLastName(): String
getAddress(): Address
getEmails(): List<String>
getFullname(): String
getPersonId(): int
getEmail(): String
toString(): String
Iterable
T
SortedList
T
info: T[]
measure: int
comparator: Comparator<T>
SortedList(comparator: Comparator<T>)
add(element: T): void
remove(item: T): boolean
size(): int
get(index: int): T
iterator(): Iterator<T>
InvoiceComparator
customerNameComparator(): Comparator<Invoice>
totalValueComparator(): Comparator<Invoice>
storeAndSalespersonComparator(): Comparator<Invoice>
DatabaseLoader
LOGGER: Logger
getEmailById(personId: int): List<String>
personConnectToDatabase(): Map<Integer,Person>
storeConnectToDatabase(persons: Map<Integer,Person>): Map<Integer,Store>
invoiceConnectToDatabase(mapStore: Map<Integer,Store>, mapPerson: Map<Integer,Person>): Map<Integer,Invoice>
itemConnectToDatabase(): Map<Integer,Item>
invoiceItemsConnectToDatabase(items: Map<Integer,Item>, invoices: Map<Integer,Invoice>): Map<Integer,Invoice>

2.1. Component Testing Strategy

To ensure the effectiveness and reliability of our Invoice Reporting System, a comprehensive testing strategy was implemented for each component. Initially, we used CSV files containing data for Items, Persons, Stores, InvoiceItems, and Invoices to test the Java classes responsible for representing and processing these entities. By using these CSV files as test data, we were able to validate the functionality of our Java classes and identify any issues or inconsistencies in the data handling process. This testing approach allowed us to verify that each class could accurately parse, process, and store the information contained in the respective CSV files.

3. Database Interface

The database interface was designed to enable Invoice Reporting System to interact with the database. The interface was implemented using JDBC (Java Database Connectivity) to ensure that the system could communicate with any standard SQL database. The database was designed to store the different entities defined in the class/entity model above. The database was normalized to ensure data consistency and integrity.

The API allowed the application to connect to the database and perform CRUD (Create, Read, Update, Delete) operations. The API included data validation and error-handling mechanisms to ensure that the data stored in the database was accurate and consistent. Additionally, the API enabled the system to perform real-time reporting and data analysis, allowing the business to make quicker decisions and respond more efficiently to market trends and customer needs.

4. Design & Integration of Data Structures

The Sorted List ADT (Abstract Data Type) was designed to enhance the application's ability to organize, manage, and analyze sales data. The Sorted List ADT provided the following benefits to the overall project:

- Efficient Data Organisation: By implementing a Sorted List ADT using a binary search tree data structure, the sales data was organized hierarchically, allowing for efficient sorting and searching operations. This structured organization of data enabled the application to easily sort sales data based on various criteria, such as by store or salesperson.

- Improved Report Generation: With the Sorted List ADT integrated into the InvoiceReport class, the application could generate summary reports more efficiently. The hierarchical organization of data enabled faster access to the required information, resulting in quicker generation of summary reports, store summaries, and invoice summaries.

- Enhanced Data Analysis: The Sorted List ADT allowed for faster data retrieval and manipulation, enabling the application to perform data analysis more efficiently. This improved data analysis capability helped FarMarT identify trends, make informed decisions, and allocate resources effectively.

- Streamlined Integration: Updating the LoaderFiles class to load data into the Sorted List ADT instead of Java objects ensured a seamless integration of the ADT into the existing application. This streamlined integration allowed the application to take advantage of the benefits provided by the Sorted List ADT without major changes to the existing codebase.

By incorporating the Sorted List ADT into the Invoice Reporting System, the overall project gained a significant improvement in its ability to process and analyze sales data effectively. This enhancement, in turn, contributed to better decision-making and improved operational efficiency for FarMarT, supporting the company's growth and success in the market.

4.1. Component Testing Strategy

The component testing strategy for the Invoice Reporting System ensured accuracy and reliability by identifying test scenarios based on project requirements, preparing representative test data, executing tests, comparing actual and expected outputs, addressing any identified issues, and documenting test results. This meticulous testing approach confirmed the proper functioning of individual components, such as the Sorted List ADT, and contributed significantly to the overall success and reliability of the system.

5. Changes & Refactoring

Throughout the development lifecycle, we made several changes and refactored the implementations to improve the design and functionality of the application. Some of the major changes and refactoring include:

- Redesigning the database schema to improve data consistency and integrity

- Refactoring the API to improve error handling and reporting

- Refactoring the Java classes to improve code readability and maintainability

2. Bibliography

[1] XML (Extensible Markup Language) - World Wide Web Consortium (W3C). (n.d.). Extensible Markup Language (XML). Retrieved from https://www.w3.org/XML/

[2] Gson. (n.d.). Gson User Guide. Retrieved from https://github.com/google/gson/blob/master/UserGuide.md