DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

# FarMarT Project

Document Design

**Cong Nguyen and Yashraj Purbey**

**05/12/2023**
**Document Version 6.0**

## Revision History

| Version | Description of Change(s) | Author(s) | Date |
|---|---|---|---|
| 1.0 | Initial draft of the design document | Cong Nguyen and Yashraj Purbey | 2023/02/17 |
| 2.0 | Second draft of the design document | Cong Nguyen and Yashraj Purbey | 2023/03/03 |
| 3.0 | Third draft of the design document | Cong Nguyen and Yashraj Purbey | 2023/03/24 |
| 4.0 | Fourth draft of the design document | Cong Nguyen and Yashraj Purbey | 2023/04/07 |
| 5.0 | Fifth draft of the design document | Cong Nguyen and Yashraj Purbey | 2023/04/21 |
| 6.0 | Finale version of the design document | Cong Nguyen and Yashraj Purbey | 2023/05/12 |

## Contents

# 1. Introduction

The main goal of this project is to design a database, and a Java system to support the business model for Pablo Myers, who is an owner of FarMarT(FMT), which is a family business sales company selling agricultural products, equipment, and related services to farmers.

## 1.1 Purpose of the Document

The purpose of the project is to help the company transfer the written-in-hand sales reports into digital ones. By doing that, it helps to enhance FMT's business operations with more technical in checking their sales activities.

## 1.2 Scope of the Document

The scope of this project is to design a program that supports FMT's business by keeping track of all sales in a database system and helping the client to create reports on the sales of farming equipment, products, and service contracts.

There are some rules related to the sales activities that the project should follow:

(a) The company's profit comes from three sold items:

- Farming equipment (which can be purchased or leased).
- Products (such as fertilizer, seed, etc.).
- Service contracts (such as plowing, harvesting, or training).

(b) The project helps the company to keep track of invoices. Each invoice includes:

- A unique identifying alphanumeric code.
- The date of creating invoice
- The customer's information
- The salesperson's information
- The store's information
- The sold item's information

(c) Each item has several rules to measure the sales price, and taxes:

- Equipment:
    - If the equipment is purchased, the customer pays the full price, and there are no taxes.
    - If the equipment is leased, the customer pays the price for every 30 days period. For leases costing less than $10,000, zero tax is applied; for leases costing $10,000 but less than $100,000 a tax of $500 is applied; for all leases $100,000 or more, a tax of $1,500 is applied.
- Product: The sales price is measured by the quantity times with price per unit. All products taxes are 7.15%.
- Service: The sales price is measured by the number of hours times with price per hour. All services taxes are 3.45%.

The goal of this project is to output the four reports supporting the sales activities:

- Summary report: Report the number of total taxes, and total sales price of each invoice.
- Store report:  Report the number of total taxes, and total sales price of each store.
- Detail report: Report in detail on each invoice.
- Sales report: Report on the sales activities based on some specific sorting orders.

### 1.3 Abbreviations & Acronyms

- FMT - Company FarMarT
- JDBC - Java Database Connectivity
- ADT - Abstract Data Types
- CSV - Comma-separated values
- XML - Extensible Markup Language
- JSON - JavaScript Object Notation
- OOP - Object-Oriented Programming
- RDBMS - Relational Database Management System

## 2. Overall Design Description

This section explains the four overall designs in the project: database design, class design, and JDBC design. These designs are used to support outputting the sales reports.

### 2.1 Database design

For the database design, the project uses the MySQL program to conduct the tables, which store the data of the sales activities.
The database contains the persons' information, stores' information, items' information, and invoices' information, which are used for the sales reports.

### 2.2 Class design

For the class design, the project uses the Java program to conduct classes. Each class has a different objective to support creating the sales reports.

### 2.3 JDBC design

For the JDBC design, the project conducts some classes in the Java program to connect the database to Java. This helps Java to load the data from database into Java or insert the data from Java to database for the purpose of creating the sales reports.

### 2.4 Alternative Design Options

Instead of connecting to the data in MySQL, the project can conduct reports based on the CSV files. These CSV files contain information about persons, stores, items, and invoices. This means Java has one class to parse the data from these csv files to create sales reports, and the project does not need to use MySQL to store the sales information.

This alternative approach still helps to conduct the sales reports, but it has some disadvantages:

- To access the database in MySQL, the user needs to have an account and password, while there is no password to access the CSV file. Thus, using CSV files to store data is less secure than MySQL.
- MySQL is a database management system. It has convenient tools to store the data or define the variables that are suitable for the rules related to sales activities.

## 3. Database Section

### 3.1 Design Instructure

The project's database runs based on the MySQL program. The database in MySQL has nine tables. It is designed based on the relationship between tables, which foreign keys are referred to other primary keys in other tables, and RDBMS principles. Some columns in some tables needed to be unique, and not null because of avoiding the duplicated and null data:

- **State:** the column stateName is defined as unique and not null
- **Country:** the column countryName is defined as unique and not null
- **Address:** The city, street, state Id, and country Id are designed as a unique group to avoid duplicated data. It has the foreign keys connecting with table State, and Country.
  - ⇨ For normalizing geographical data, the database separates the information of State, and Country to Address.
- **Person:** the column personCode is defined as unique, not null to identify each person. It has the foreign keys connecting with table Address to get the information related to Address.
- **Email:** It has the foreign keys connecting with table Person.
  - ⇨ Because one person can have multiple emails, the database separates the information of Email to Person.
- **Store:** the column storeCode is defined as unique, not null to identify each store. It has the foreign keys connecting with table Person, and Address to get the information related to Person, and Address, respectively.
  - ⇨ The column named manageId in table Store is not named personId like the primary keys in table Person because of emphasizing that is the manager of the store.
- **Item:** the column itemCode is defined as unique, not null to identify each item. There is a check condition to only allow some specific letter for column typeItem. The column typeItem only contains one letter, which are "E", "P", or "S", that are used to separate equipment, product, or service, respectively. Some columns are allowed to be null because of different items' information:
  - Equipment: column model
  - Product: column unit, and unitPrice
  - Service: column hourlyRate

- **Invoice:** the column invoiceCode is defined as unique, not null to identify each invoice. It has the foreign keys connecting with table Person, and Store to get the information related to Person, and Store, respectively.
  - ⇨ The column named customerId, and salesPersonId in table Invoice is not named personId like the primary keys in table Person because of emphasizing that is the customer, or salesperson in the invoice.
- **InvoiceItem:** It has the foreign keys connecting with table Invoice, and Item to connect the information of Item to Invoice. There is a check condition to only allow some specific letter for column typeEquipment. The column typeEquipment only contains one letter, which is “P”, or “L”, that are used to separate purchased, or lease equipment respectively. Some columns are allowed to be null because of different items’ information:
  - Equipment: column typeEquipment, price, beginLease, and endLease
  - Product: column quantity
  - Service: column amountOfHour
  - ⇨ The relationship between table Invoice and Item is a many-to-many relationship, as one invoice can have multiple items, and one item can be in multiple invoices. Thus, the table InvoiceItem is acted as the intermediate table between them.

The order of creating tables is followed: State -> Country -> Address -> Person -> Email -> Store -> Invoice -> Item -> InvoiceItem, because of foreign keys rule. The order of dropping tables is reversed from the creating order.

## 3.2 Test Cases

The project has one test case. To test whether each table stored the data correctly, the project inserts some records into the database. For easier to identify the records, the project follows some rules to create data for these tables:

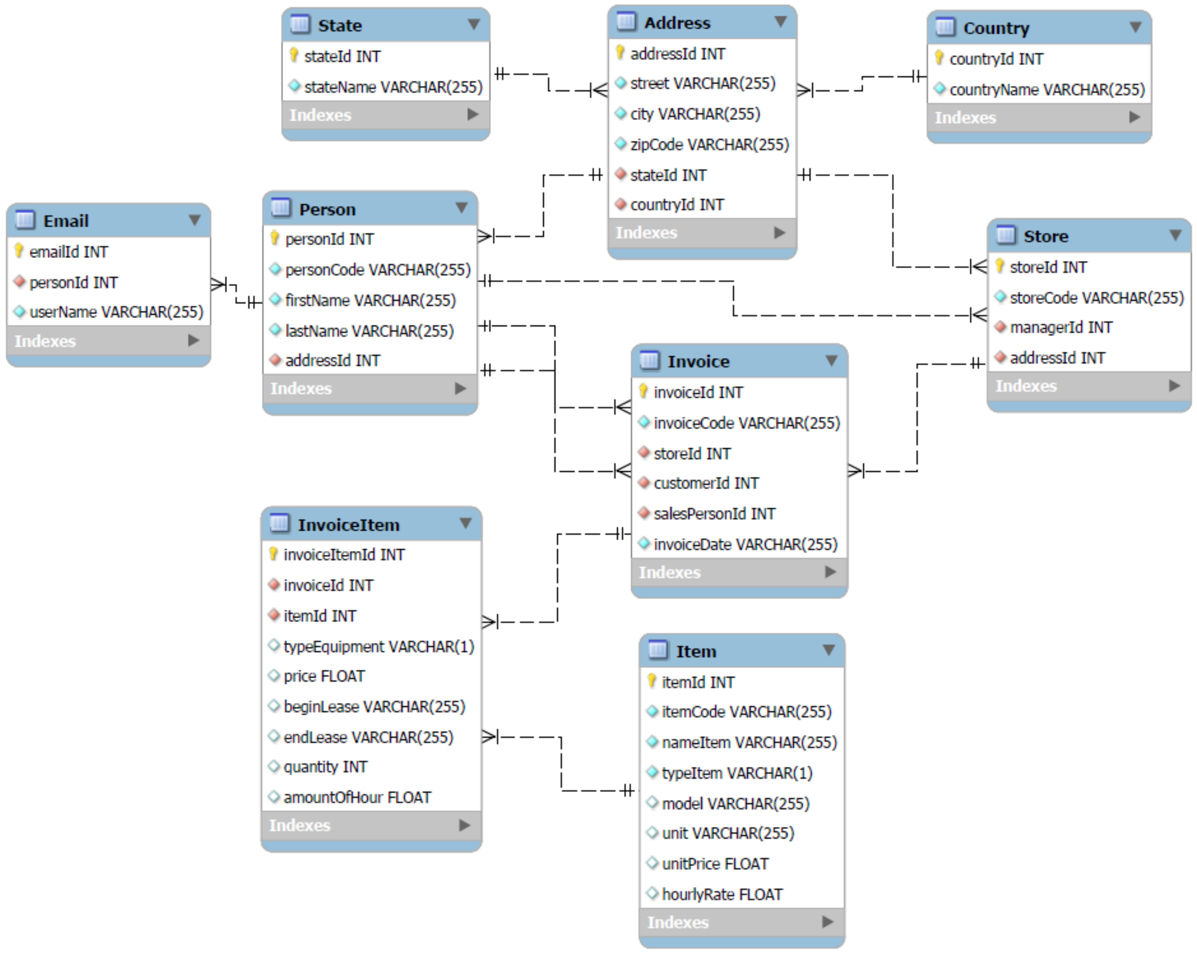
- pesonCode: it starts with SAL for manager, and salesperson, and PUR for customer. After that, it continues with the ascending numerical order. For example: SAL001 for the first manager, or the salesperson, while PUR001 for the first customer.
- storeCode: it starts with TITS, and after that, it continues with the ascending numerical order. For example: TITS001 for the first store.
- invoiceCode: it starts with BIL, and after that, it continues with the ascending numerical order. For example: BIL001 for the first invoice.
- itemCode: it starts with IE for equipment, IP for product and IS for service. After that, it continues with the ascending numerical order. For example: IE0001 for the first equipment, IP0001 for the product, while IS0001 for the first service.

To easily manage the testing records, the database has two tabs: one is to create tables and insert data, and one is to execute testing queries. While inserting data for testing, the project separates each kind of record based on the table’s information.

Besides, to test the ability of a database to delete the record of one person, the project comes up with the soft delete by conducting one more column in table Person to check the status, whether deleted or not, of that person. The reason for using soft delete is to avoid the hard deleting of the related data to this person. For example, in case of deleting a manager, customer or a salesperson's information, the database cannot delete their column id, as they have foreign keys related to table Invoice, or Store.

Also, the project conducts a query to detect a potential instance of fraud where an employee makes a sale to themselves.

## 3.3 ER Diagram



## 4. Class Section

### 4.1 Class Design

This project is runs based on a Java program. The main class, which has the name InvoiceReport is used for printing out the sales reports.

There are 25 classes created in Java. Each of them is designed based on the OOP principles. Below is the explanation of creating each class:

- **Person:** Contain the information of manager, salesperson, and customer. Because one person has many emails, the project uses a List to store the information of all emails of one person.
- **Address:** Contain the information of address belonging to a person, or a store.
- **Store:** Contain the information of all stores. Because one store has many invoices, the project has one variable as a List to store the information of all invoices. This means that based on the list of invoices in the store, the program can get all information that relates to the invoices.
  - ⇨ Although, the person's information and store's information have the address, the project choose to separate the class Address to Person, and Store due to objective purpose. The reason is that the information of street, city, zip code, state, and country are belonged to class Address, not Person, or Store.
- **Item:** Contain the information of sales item. This class is defined as an abstract class because an abstract class allows the project to create methods that its subclasses can implement or override. There are three abstract methods in this class, which are getSubtotal(), getTaxes(), and getGrandTotal() to allow the overriding in the subclasses. Three subclasses are:
  - **Equipment:** This class also has two subclasses, which are for **EquipmentPurchase** and **EquipmentLease**. Although class Equipment has two subclasses, the project does not define this class as abstract class, because it needs to use the constructor in class Equipment for parsing data from CSV files.
  - **Product:** the variable taxRateProduct is defined as private final static because it is a constant.
  - **Service:** the variable taxRateService is defined as private final static because it is a constant.
  - ⇨ In each subclasses, there are three override methods, which are getSubtotal(), getTaxes(), and getGrandTotal(), that help to override the abstract methods in class Item.
- ⇨ In the class Person, Address, Store, and Item, there are copy constructors that are used for overloading. Each copy constructor conducts the new constructor based on the old constructor, with one new variable. The new variable is the id, which is used to store the id when the project loads the data from database during JDBC.
- **Invoice:** Contain the information of invoice. Because one invoice has many invoices, the project has one variable as a List to store the information of all sales items. This means based on the list of items in the invoice, the program can get all information that relates to the items.

⇨ All the variables in each class mentioned above are defined as private, and final. The reason is to avoid manual changing the value of these variables. This means that the only way to change the value is to access the database in MySQL, or CSV files.

- **ParsingCsvFile:** Parse the data from CSV files. There are five CSV files: Person.csv, Store.csv, Item.csv, Invoice.csv, and InvoiceItem.csv. Each record in each CSV file is identified based on personCode, storeCode, itemCode, and InvoiceCode, respectively.

  ⇨ While parsing data, the data is parsed as a HashMap, because of matching the key, which is the code, to the value of each record. By doing this, it helps to keep track of the data during parsing. However, while parsing the data from InvoiceItem.csv, the data is parsed as a List because one invoice can contain multiple items, as if using the HashMap, it violated the rule of one unique key mapping to one value.

- **DataConverterUtils:** Contain some helper methods to convert a HashMap to a List, or creating XML, and JSON file.

  ⇨ **GsonExclusiveField**, and **GsonOmitField:** Both classes contain methods to exclude some variables during converting to JSON file. This is for security purpose, due to avoid transferring data while creating JSON files. For XML files, the project uses *@XStreamOmitField* on the top of the chosen variables to avoid transferring data.

- **DatabaseLoader:** Load the data from MySQL. It has a helper class **DatabaseLoaderUtils.**

- **DatabaseConnection:** Contain some methods factoring the connection to database in MySQL. It is a helper method to open and close connections to the database. It has a helper class **DatabaseInfo**, which contains information of username, password, and URL address.

- **InvoiceData:** Insert the data from Java into MySQL. It has a helper class **InvoiceDataUtils.**

- **SortedLinkedList:** A linked list ADT to portray the functionality of one linked list. It has a helper class **Node**.

⇨ Some classes have helper classes to avoid repeating the same code in different code again and again.

- **ReportFormating:** Contain many methods to format reports.
- **MainMethods:** Contain many methods for each phase of the project. The purpose of this class is to test the results of each phase.
- **InvoiceReport:** Contain the main method to execute all methods in class MainMethods to output the sales reports.

## 4.2 Test Cases

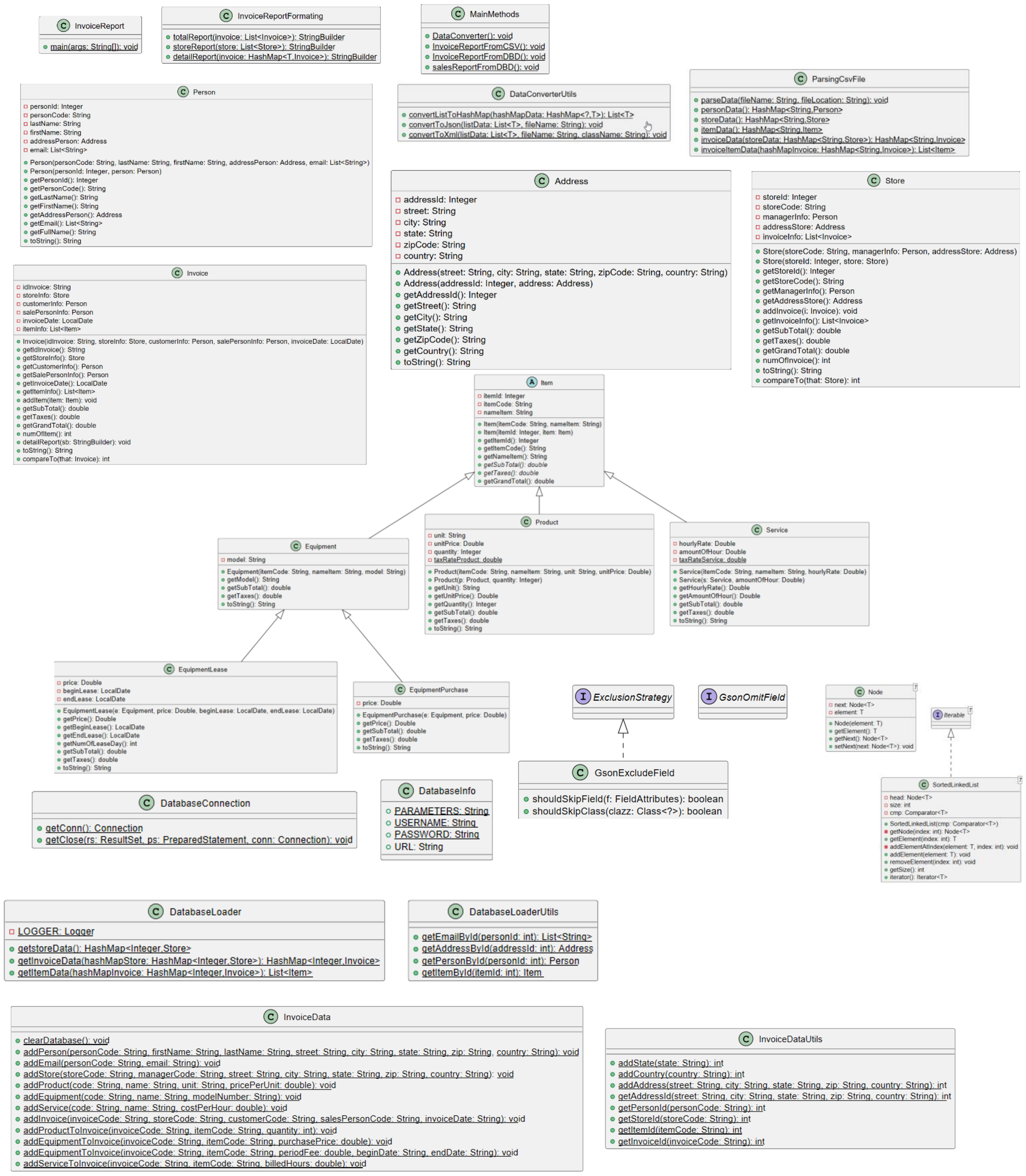
In this section, the project focuses on testing data from CSV files.

The program has several test cases coming from a CSV file. For testing purposes, on the first line, the file shows the number of records. Each record has information separated by a comma ",". The order of each CSV file must be name exactly, and followed as shown below:

- Persons.csv: person code, last name, first name, street, city, state, zip code, country, list of email
- Store.csv: store code, manager code, street, city, state, zip code, country
- Item.csv:
    - To test the equipment: item code, E, item name, model
    - To test the product: item code, P, item name, unit, unit price
    - To test the service: item code, S, item name, hourly payment
- Invoices.csv: invoice code, store code, customer code, salesperson code, date of invoice (yyyy-mm-dd)
- InvoiceItem.csv:
  - To test the equipment:
    - For purchasing: invoice code, item code, P, price.
    - For leasing: invoice code, item code, L, price, beginning lease date, ending lease date (yyyy-mm-dd, which means year-month-day).
  - To test the product: invoice code, item code, quantity
  - To test the service: invoice code, item code, amount of hour

While creating the CSV file, the person code, store code, item code, invoice code still follows the rule as mentioned in the test case of Database section. This helps to keep track and identify the data.

## 4.3 UML Diagram


InvoiceReport
main(args: String[]): void
InvoiceReportFormating
totalReport(invoice: List<Invoice>): StringBuilder
storeReport(store: List<Store>): StringBuilder
detailReport(invoice: HashMap<T,Invoice>): StringBuilder
MainMethods
DataConverter(): void
InvoiceReportFromCSV(): void
InvoiceReportFromDBD(): void
salesReportFromDBD(): void
DataConverterUtils
convertListToHashMap(hashMapData: HashMap<?,T>): List<T>
convertToJson(listData: List<T>, fileName: String): void
convertToXml(listData: List<T>, fileName: String, className: String): void
ParsingCsvFile
parseData(fileName: String, fileLocation: String): void
personData(): HashMap<String,Person>
storeData(): HashMap<String,Store>
itemData(): HashMap<String,Item>
invoiceData(storeData: HashMap<String,Store>): HashMap<String,Invoice>
invoiceItemData(hashMapInvoice: HashMap<String,Invoice>): List<Item>
Person
personId: Integer
personCode: String
lastName: String
firstName: String
addressPerson: Address
email: List<String>
Person(personCode: String, lastName: String, firstName: String, addressPerson: Address, email: List<String>)
Person(personId: Integer, person: Person)
getPersonId(): Integer
getPersonCode(): String
getLastName(): String
getFirstName(): String
getAddressPerson(): Address
getEmail(): List<String>
getFullName(): String
toString(): String
Address
addressId: Integer
street: String
city: String
state: String
zipCode: String
country: String
Address(street: String, city: String, state: String, zipCode: String, country: String)
Address(addressId: Integer, address: Address)
getAddressId(): Integer
getStreet(): String
getCity(): String
getState(): String
getZipCode(): String
getCountry(): String
toString(): String
Store
storeId: Integer
storeCode: String
managerInfo: Person
addressStore: Address
invoiceInfo: List<Invoice>
Store(storeCode: String, managerInfo: Person, addressStore: Address)
Store(storeId: Integer, store: Store)
getStoreId(): Integer
getStoreCode(): String
getManagerInfo(): Person
getAddressStore(): Address
addInvoice(i: Invoice): void
getInvoiceInfo(): List<Invoice>
getSubTotal(): double
getTaxes(): double
getGrandTotal(): double
numOfInvoice(): int
toString(): String
compareTo(that: Store): int
Invoice
idInvoice: String
storeInfo: Store
customerInfo: Person
salePersonInfo: Person
invoiceDate: LocalDate
itemInfo: List<Item>
Invoice(idInvoice: String, storeInfo: Store, customerInfo: Person, salePersonInfo: Person, invoiceDate: LocalDate)
getIdInvoice(): String
getStoreInfo(): Store
getCustomerInfo(): Person
getSalePersonInfo(): Person
getInvoiceDate(): LocalDate
getItemInfo(): List<Item>
addItem(item: Item): void
getSubTotal(): double
getTaxes(): double
getGrandTotal(): double
numOfItem(): int
detailReport(sb: StringBuilder): void
toString(): String
compareTo(that: Invoice): int
Item
itemId: Integer
itemCode: String
nameItem: String
Item(itemCode: String, nameItem: String)
Item(itemId: Integer, item: Item)
getItemId(): Integer
getItemCode(): String
getNameItem(): String
getSubTotal(): double
getTaxes(): double
getGrandTotal(): double
Equipment
model: String
Equipment(itemCode: String, nameItem: String, model: String)
getModel(): String
getSubTotal(): double
getTaxes(): double
toString(): String
Product
unit: String
unitPrice: Double
quantity: Integer
taxRateProduct: double
Product(itemCode: String, nameItem: String, unit: String, unitPrice: Double)
Product(p: Product, quantity: Integer)
getUnit(): String
getUnitPrice(): Double
getQuantity(): Integer
getSubTotal(): double
getTaxes(): double
toString(): String
Service
hourlyRate: Double
amountOfHour: Double
taxRateService: double
Service(itemCode: String, nameItem: String, hourlyRate: Double)
Service(s: Service, amountOfHour: Double)
getHourlyRate(): Double
getAmountOfHour(): Double
getSubTotal(): double
getTaxes(): double
toString(): String
EquipmentLease
price: Double
beginLease: LocalDate
endLease: LocalDate
EquipmentLease(e: Equipment, price: Double, beginLease: LocalDate, endLease: LocalDate)
getPrice(): Double
getBeginLease(): LocalDate
getEndLease(): LocalDate
getNumOfLeaseDay(): int
getSubTotal(): double
getTaxes(): double
toString(): String
EquipmentPurchase
price: Double
EquipmentPurchase(e: Equipment, price: Double)
getPrice(): Double
getSubTotal(): double
getTaxes(): double
toString(): String
ExclusionStrategy
GsonOmitField
GsonExcludeField
shouldSkipField(f: FieldAttributes): boolean
shouldSkipClass(clazz: Class<?>): boolean
Node
T
next: Node<T>
element: T
Node(element: T)
getElement(): T
getNext(): Node<T>
setNext(next: Node<T>): void
Iterable
T
SortedLinkedList
T
head: Node<T>
size: int
cmp: Comparator<T>
SortedLinkedList(cmp: Comparator<T>)
getNode(index: int): Node<T>
getElement(index: int): T
addElementAtIndex(element: T, index: int): void
addElement(element: T): void
removeElement(index: int): void
getSize(): int
iterator(): Iterator<T>
DatabaseConnection
getConn(): Connection
getClose(rs: ResultSet, ps: PreparedStatement, conn: Connection): void
DatabaseInfo
PARAMETERS: String
USERNAME: String
PASSWORD: String
URL: String
DatabaseLoader
LOGGER: Logger
getstoreData(): HashMap<Integer,Store>
getInvoiceData(hashMapStore: HashMap<Integer,Store>): HashMap<Integer,Invoice>
getItemData(hashMapInvoice: HashMap<Integer,Invoice>): List<Item>
DatabaseLoaderUtils
getEmailById(personId: int): List<String>
getAddressById(addressId: int): Address
getPersonById(personId: int): Person
getItemById(itemId: int): Item
InvoiceData
clearDatabase(): void
addPerson(personCode: String, firstName: String, lastName: String, street: String, city: String, state: String, zip: String, country: String): void
addEmail(personCode: String, email: String): void
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
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
InvoiceDataUtils
addState(state: String): int
addCountry(country: String): int
addAddress(street: String, city: String, state: String, zip: String, country: String): int
getAddressId(street: String, city: String, state: String, zip: String, country: String): int
getPersonId(personCode: String): int
getStoreId(storeCode: String): int
getItemId(itemCode: String): int
getInvoiceId(invoiceCode: String): int

## 5. Database Connectivity Section

This section focuses on explaining why the project uses the JDBC to connect the database from MySQL to Java, and the test case.

### 5.1 Design explanations

There are 6 classes in Java that are used to explain how JDBC works, which are DatabaseConnection, DatabaseInfo, DatabaseLoader, DatabaseLoaderUtils, InvoiceData, and InvoiceDataUtils. Each class has different functionality that is already described in the Class section.

To track the connection between the database and the code in Java, the type of variables in each table of database are matched with the variables of these classes:

- Table Person, and Email with class Person.
- Table Address, State, and Country with class Address.
- Table Store with class Store.
- Table Item with class Equipment, Product, and Service.
- Table Invoice with class Invoice.
- Table InvoiceItem with class Invoice, EquipmentPurchase, EquipmentLease, Product, and Service.

To load the data from database, Java uses two classes: DatabaseLoaderUtils, and DatabaseLoader. DatabaseLoaderUtils contains helper methods to load the data based on the primary keys in the tables, that supports the methods in DatabaseLoader. In the class DatabaseLoader, three methods help to load the data to support for creating sales reports:

- The method **getStoreData()** creates a HashMap<Integer, Store> that maps the store Id to its information related to the store.
- The method **getInvoiceData()** creates a HashMap<Integer, Invoice> that maps the invoice Id to its information related to the invoice. This method has a parameter of HashMap<Integer, Store> to load all data from the getStoreData() into this method.
- The method **getItemData()** creates a List of item that is used in the invoice. This method has a parameter of HashMap<Integer, Invoice> to load all data from the getInvoiceData() into this method. The reason to choose a List, instead of HashMap is that an invoice can obtain many items, which violates the rule of unique key mapping to one value.

⇨ When Java calls these three methods in the main program, this helps to create a list of invoices that contain all information of the customer, salesperson, store, and sales item.

To add the data into database, Java uses two classes: InvoiceData, and InvoiceDataUtils. Class InvoiceDataUtils has some helper methods to check the existing data before adding it to the database. This helps to avoid duplicated data in the database. Class InvoiceData contains methods to add data into database. Besides, it also has the method to clear all records in the database to make all tables in the database fresh. Each adding methods in InvoiceData inserts the records as follows:

- addPerson() to table Person

- addEmail() to table Email
- addStore() to table Store
- addProduct() to table Item containing only records of Product
- addEquipment() to table Item containing only records of Equipment
- addService() to table Item containing only records of Service
- addInvoice() to table Invoice
- addProductToInvoice() to table InvoiceItem containing only records of Product
- addEquipmentToInvoice() to table InvoiceItem containing only records of Equipment
- addServiceToInvoice(() to table InvoiceItem containing only records of Service

⇨ This helps to keeps track of the records between each Java class and each table in the database.

### 5.2 Test cases

There are two kinds of test cases: loading data from database and add data to a database in MySQL.

For loading the data from the database, the test case comes from the database of MySQL. To check the consistency of the result in the three reports, the project compares the value between the output reports from the CSV file and the one from the database. This means the inserted data from the database must be the same as the CSV file.

For inserting data to the database, the test case produces some tests to check the duplicated data, and invalid data type of each variable.

The program also produces a test case to check the valid username, password, and URL.

## 6. Data Structure Section

This section focuses on explaining why the project uses ADT to create a sorted linked list to create sales reports.

### 6.1 Design explanations

Two classes in Java that are used to explain how JDBC works, which are Node, and SortedLinkedList. The project chooses the linked list as it helps to support the business model of FMT. The node holds both the data and the reference to the next node in the list.

In the class SortedLinkedList, it contains methods to add, get, and remove elements to support creating a linked list. Some methods such as **getNode()**, or **addElementAtIndex()**, which are the helper method, are defined as private, because of avoiding manipulating the order of sorted list data. For example, to add the data into the list, the program only allows the users to add the value, but not the position of the value in the list. The program automatically does the sorting for the users, so it can avoid the users manipulating the order of the sorted list. Besides, by designing to sort the data while adding the data, instead of adding the data, then sorting the data, it helps to shorten the process.

## 6.2 Test case

The test case comes from the database. The purpose of the test case is to check whether the created sorted list follows the order of the sales report' requirements. These sales reports have a sorted order showing as below:

- Last name/ first name of the customer in alphabetic order
- Descending order for the total value of the invoice
- An ordering that groups all invoices by their store, and then by the salesperson by last name/ first name.

Also, the project has a test case to assess the ability to add, remove, get the position of elements, and get size of the linked list.