Sales Subsystem

Design Document

Matthew Bachmann
matthew.bachmann@huskers.unl.edu
University of Nebraska—Lincoln

May 12, 2023
Version 6.0

A technical design document for a database backed Java application implementing a sales subsystem for FarMarT regional framing chain.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Matthew Bachmann	2023/02/16
2.0	Updated UML diagram	Matthew Bachmann	2023/03/03
3.0	Updated Database Design	Matthew Bachmann	2023/03/23
4.0	Added Database API	Matthew Bachmann	2023/04/07
5.0	Responding to feedback, updating diagrams	Matthew Bachmann	2023/04/07
6.0	Data Structure Integration	Matthew Bachmann	2023/05/12

Contents

1 Introduction

This document outlines the technical design for a data backed Java application for the FarMarT regional chain managed by Pablo Myers. FarMarT sells farm equipment and supplies to farmers. The company is updating all business operations with newly developed systems for inventory, marketing, delivery, and sales. This document describes the design of the sales subsystem.

The sales subsystem manages invoices for the sale of various items including farming equipment, products, and contracts for services. Each type of item has unique costs associated with it:

- **Equipment** can be purchased or leased. For leased items there is an associated 30-day cost prorated based on the number of days leased. No taxes are assessed on the purchase of equipment. Leased equipment is charged a flat tax based on the price of the lease.
- **Products** are sold in quantities by a specified unit. Each unit has a particular price/unit and cannot be purchased in fractions of the unit. A tax rate of 7.15% is assessed to all sales of products.
- **Services** can be contracted with FarMarT and are billed on a per-hour basis (which can be fractional). A tax rate of 3.45% is assessed for all services.

The main functionality goals of the subsystem are to:

G1. Manage the data listed above in an organized and effective way.

G2. Provide functionality to edit the data.

G3. Serialize the data in a way that can easily be shared.

G4. Generate various sales reports.

1.1 Purpose of this Document

The purpose of this document is to describe the design of the sales subsystem and provide an overview of its implementation.

1.2 Scope of the Project

This sales system is designed in specifications in line with the request of FarMarT which are described above. The main goal is to provide a sales subsystem that meets the business needs of FarMarT.

Since other systems are being developed (inventory, marketing, and delivery) all data needs to be serialized into a format that represents the basic objects of our system.

The application therefore provides functionality to output data files in either XML[3] or JSON [1] format (see **G3** above).

The sales subsystem also provides functionality to generate the following three reports (see **G4** above):

1. Summary of all sales.
2. Another summary of all sales but for specific stores.
3. Individual invoices.

To increase ease of data management (see **G1** above), the data provided by FarMarT is added to a SQL database. The application uses JDBC to connect to the database and load data into the Java objects. The sales subsystem can load data either from CSV files or from the database. The database has functionality to persist or modify data in the database (see **G2** above). Details regarding the database design can be found below.

This project is not meant to be a sales subsystem for any other company than FarMarT and the specific items sold by FarMarT.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Abbreviations & Acronyms

API	Application Programming Interface
CSV	Comma Separated Values
ER	Entity Relation
JDBC	Java Database Connectivity API
JSON	JavaScript Object Notation
FMT	FarMarT
SQL	Structured Query Language
XML	Extensible Markup Language

2 Overall Design Description

The overall design is motivated by the data provided by FarMarT. Initial data is provided in the form of CSV files with formatting convention specified by FarMarT. All files begin with the number of data entries in the file and the subsequent lines adhere to the following formatting conventions:

- **Person data** contains a list of people associated to the company: customers, managers, and salespeople. Each data entry consists of a unique alphanumeric *person code*, a *name* given in last name, first name format, an *address* with street, city, state, zip code, and country, and an optional list of email addresses. An example line of data is:

  ```
  fdc267,Burr,Rob,6 A St.,Troy,TX,31405,US,brob@gmail.com
  ```

- **Store data** contains a list of stores. Each data entry consists of a unique alphanumeric *store code*, a unique alphanumeric *manager code* which will correspond to a unique *person code* in the person data, and an *address* with street, city, state, zip code, and country. An example line of data is:

  ```
  3c0234,fdc267,2 B St.,Troy,TX,31405,US
  ```

  Notice that the manager of this store is Rob Burr from the person data example.

- **Item data** contains a list of items offered by FarMarT. Depending on the type of item listed, each data entry contains different information. All data entries have a unique alphanumeric *item code*, a *type* given as E for equipment, P for a product, and S for a service, and then the *name* of the item.

  - **Equipment data** then lists the equipment *model.* An example line of equipment data is:

    ```
    1d4d89,E,Tractor,MegaTract 150HP
    ```

  - **Product data** then lists the *unit* the product is sold in and then the *unit price.* An example line of product data is:

    ```
    342foa3,P,Haybale,bale,500
    ```

  - **Service data** then lists the *hourly rate* for the service. An example line of product data is:

    ```
    2334b23,S,Delivery,100
    ```

- **Invoice data** contains a list of invoices. Each data entry consists of a unique alphanumeric *invoice code*, the store where the sale occurred listed as an alphanumeric *store code* corresponding to a store in store data, the customer listed as an alphanumeric *customer code* corresponding to a person in the person data, the salesperson listed as an alphanumeric *salesperson code* corresponding to a person in the person data, and a *date* for when the sale occurred. An example line of invoice data is:

  ```
  INV001,3c0234,fdc267,8197d3,2022-12-01
  ```

- **Invoice item data** contains a list of items that have been purchased customers. Each data entry consists of a unique alphanumeric *invoice code* corresponding to an invoice in the invoice data, the item on the invoice listed as an alphanumeric *item code* corresponding to an item in the item data, and the remaining data depends on the type of item in question.

- **Equipment invoice items** then list *P* or *L* for whether the equipment was purchased or leased. If it was purchased the *price* is listed next. An example line of purchased equipment data is:

  ```
  INV001,1d4d89,P,85000
  ```

  If the equipment was leased, the 30 day *price* is listed next, and then the *start date* and *end date* are listed. An example line of leased equipment data is:

  ```
  INV001,3506f6,L,3500,2022-01-01,2022-12-31
  ```

- **Product invoice item** then lists the *quantity* purchased. An example line of purchased product data is:

  ```
  INV001,n3453js,158
  ```

- **Service invoice item** then lists the *number of hours* billed for the service. An example line of invoice service data is:

  ```
  INV001,n43k2l3,3.5
  ```

Based on the data basic Java objects are created to model:

- People (customers, salespeople, managers)
- Stores
- Addresses (both for people and store location)
- Items sold by FarMarT
- Invoices that record all information relevant to a particular purchase

Because there is a nice breakdown into basic objects, this subsystem uses an object-oriented approach.

The two major components of this project are the Java application and the SQL database. An overview of each component is given here and more details can be found in subsequent sections.

For each basic object listed above a Java class is used to model that entity in the system. The Item class is abstract since any item is necessarily a product, service, or equipment. The item class implements methods to get the cost of an item and the taxes associated to it. Java classes are implemented as subclasses of the Item class to represent products, services, and equipment. Each of these subclasses have two types of constructors. One type of constructor creates a product, service, or equipment instance that has not yet been sold to a customer. There is another constructor in each class that creates an instance of that item along with invoice data for that item. This models the instance of an item that has been purchased. The equipment class is further specified by a subclass representing leased equipment. The main constructors for each object instance are as follows:

- A product instance consisting of the *item code*, *name* (both from the super class item). In addition, the instance also has *units* the product is sold in, and the *price per unit.*
- A product instance consisting of a *product* as described above, and in addition, the *quantity* of the product purchased.
- A service instance consisting of the *item code*, *name* (both from the super class item). In addition, the instance also has *hourly rate* charged for the service.
- A service instance consisting of a *service* as described above, and in addition, the *number of hours* the service has been commissioned for.
- An equipment instance consisting of the *item code*, *name* (both from the super class item). In addition, the instance also has the *model.*
- An equipment instance consisting of an *equipment* as described above, and in addition, the *price* charged for the piece of equipment. If the piece of equipment was purchased, this is the purchase price. If the equipment was leased, this is the price per day to lease the item.
- An lease instance consisting of an *equipment* as described above, and in addition, the *start date* and *end date* for the lease.

An invoice class records all relevant information and contains a list of items that have been purchased on it. Further details regarding the class design can be found below.

The SQL database implements tables similar to the objects described above. The main difference is that many items can appear on many invoices. To implement this many to many relationship an intermediary table of invoice items is created representing items that have been purchased. Now one invoice can have many invoice items on it and one item can appear as many different invoice items.

Since each person and store has an address, both need a foreign key to access the address. Since each email is associated to a person, the email table needs a foreign key to identify the person. Each store has a foreign key to identify the manager in the person table. The invoice table has many foreign keys to access the store the sale happened at, the customer making the purchase, and the sales person. An invoice item needs a foreign key to the Invoice table to identify the invoice it appears on and a foreign key to the Item table to identify what item is being purchased.

3 Detailed Component Description

This section describes the components of the project.

3.1 Database Design

The database is designed to model the relationships between data provided by FarMarT. Please refer to Figure 1 to see an ER diagram describing the database. Each of the following relationships are one to many relations in the database and for the following reasons:

Address to Person Many people may live at the same address.

Person to Email One person may have many emails.

Person to Store One person can manage many stores.

Person to Invoice One person can be a customer or salesperson on may invoices.

Store to Invoice One store can have many invoices that occurred there.

InvoiceItem to Item One item can appear as many different invoice items.

InvoiceItem to Invoice One invoice may have many invoice items on it.

It is important to note that Item and Invoice have a many to many relationship and the intermediate table InvoiceItem serves to facilitate this relationship. Also note that in the Person to Invoice relationship, if both the sales person and customer are the same person this may be an indication of fraudulent activity. The database returns a warning if such an instance occurs.

People should only be deleted from the database if that person is not a customer or sales person on any invoice and is not a manager. The goal of this design is to keep record of employees even if they are no longer employees. Note that there is no foreign key in the Address table and so the deletion of a person does not necessitate the deletion of their address.

Furthermore, instances of the same product should not appear on a single invoice as their quantities can be combined into a single line item in an invoice. The database is designed to prevent such instances.

3.1.1 Component Testing Strategy

Test data is hard coded into the database. The test data is comprised of entries in each table represented in the Figure 1. A list of queries is written to test the integrity of the database. To ensure each table is working independently a query is used to retrieve the main attributes of each table. To ensure foreign keys are behaving correctly a query is

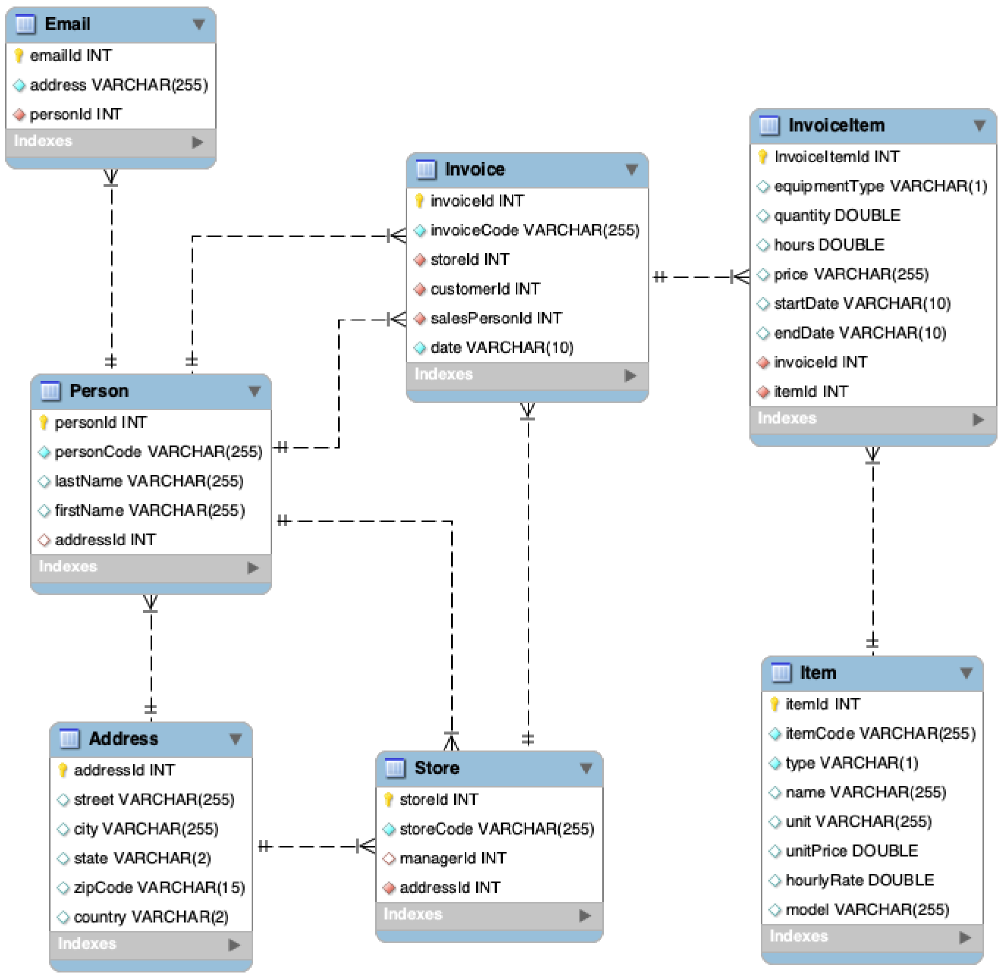
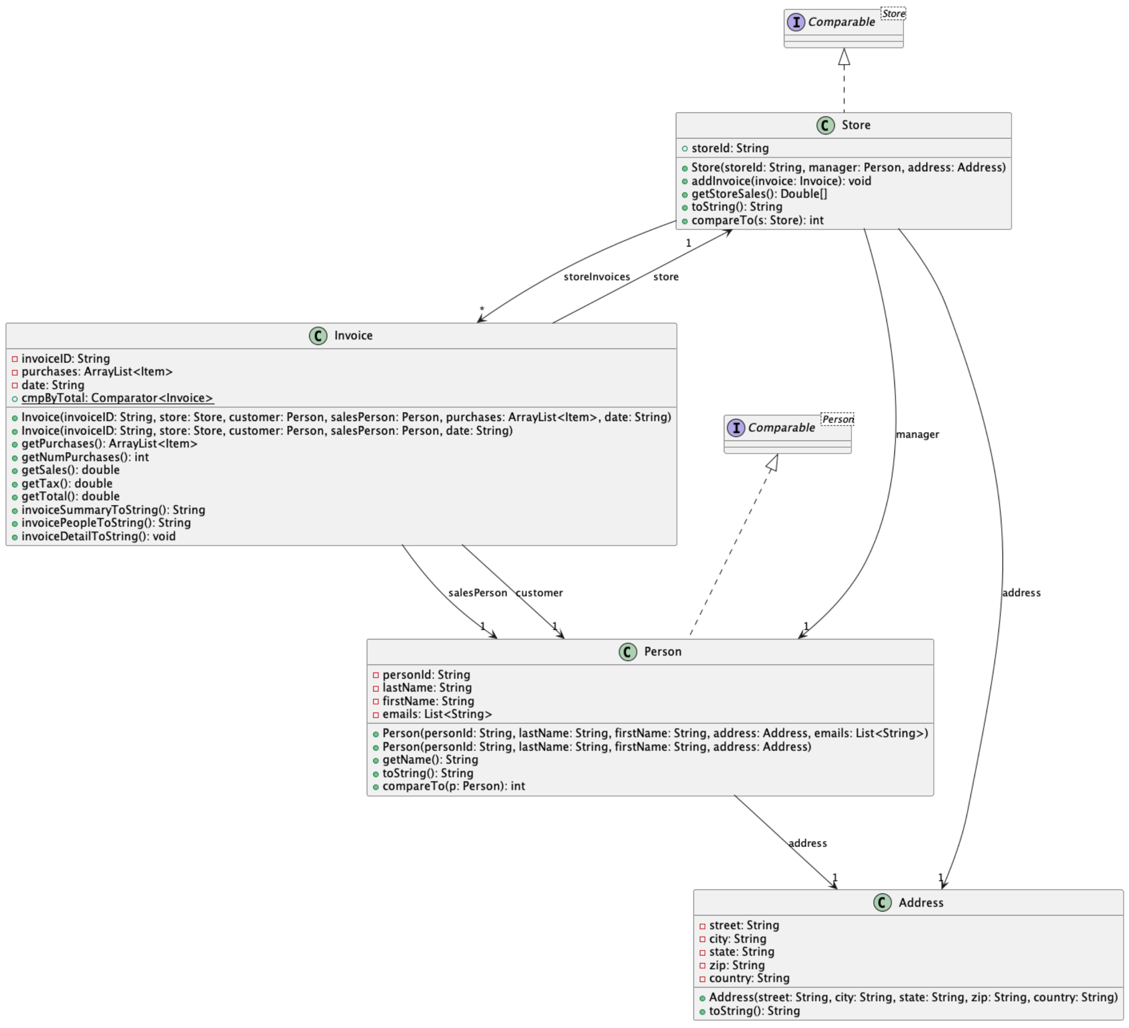


Figure 1: ER diagram for sales database

used to retrieve the major fields for tables along with other data. For example, a query is used to retrieve the major fields for every person including their address (but excluding emails). To ensure that data can be modified while maintaining data integrity a query is used to delete a person and all data corresponding to the person. To prevent fraud a query is used to see if there are invoices where the sales person and the customer are the same person. The database should issue a warning if such an instance occurs.

3.2 Class/Entity Model

The classes in Figure 3 represent the primary objects sold by FMT. The Item class is abstract because an item can only exist as a piece of equipment, a product, or a service. The Lease class is a subclass of Equipment because the start date and end date of the lease further specify the equipment. If a piece of equipment is purchased, only an Equipment instance is created and the price corresponds to the price of the purchase. For a lease, the price is the daily lease rate. Multiple constructors exist in each class: one constructor generates an instance of the item as an item offered by FMT (not yet purchased or leased), the other generates an instance of the item that has been purchased along with the associated costs.



Comparable Store
Store
storeId: String
Store(storeId: String, manager: Person, address: Address)
addInvoice(invoice: Invoice): void
getStoreSales(): Double[]
toString(): String
compareTo(s: Store): int
storeInvoices
store
1
*
Invoice
invoiceID: String
purchases: ArrayList<Item>
date: String
cmpByTotal: Comparator<Invoice>
Invoice(invoiceID: String, store: Store, customer: Person, salesPerson: Person, purchases: ArrayList<Item>, date: String)
Invoice(invoiceID: String, store: Store, customer: Person, salesPerson: Person, date: String)
getPurchases(): ArrayList<Item>
getNumPurchases(): int
getSales(): double
getTax(): double
getTotal(): double
invoiceSummaryToString(): String
invoicePeopleToString(): String
invoiceDetailToString(): void
Comparable Person
manager
address
salesPerson
customer
1
1
1
Person
personId: String
lastName: String
firstName: String
emails: List<String>
Person(personId: String, lastName: String, firstName: String, address: Address, emails: List<String>)
Person(personId: String, lastName: String, firstName: String, address: Address)
getName(): String
toString(): String
compareTo(p: Person): int
address
1
1
Address
street: String
city: String
state: String
zip: String
country: String
Address(street: String, city: String, state: String, zip: String, country: String)
toString(): String

Figure 2: UML diagram for additional classes

Figure 2 shows the person, store, and address class and how these classes are related to an invoice.

3.2.1 Component Testing Strategy

Each major component of the class model has testing implemented.

- **Data Loading** Data is loaded from CSV files with a format specified by FMT described above. All data is assumed to be correctly formatted as per

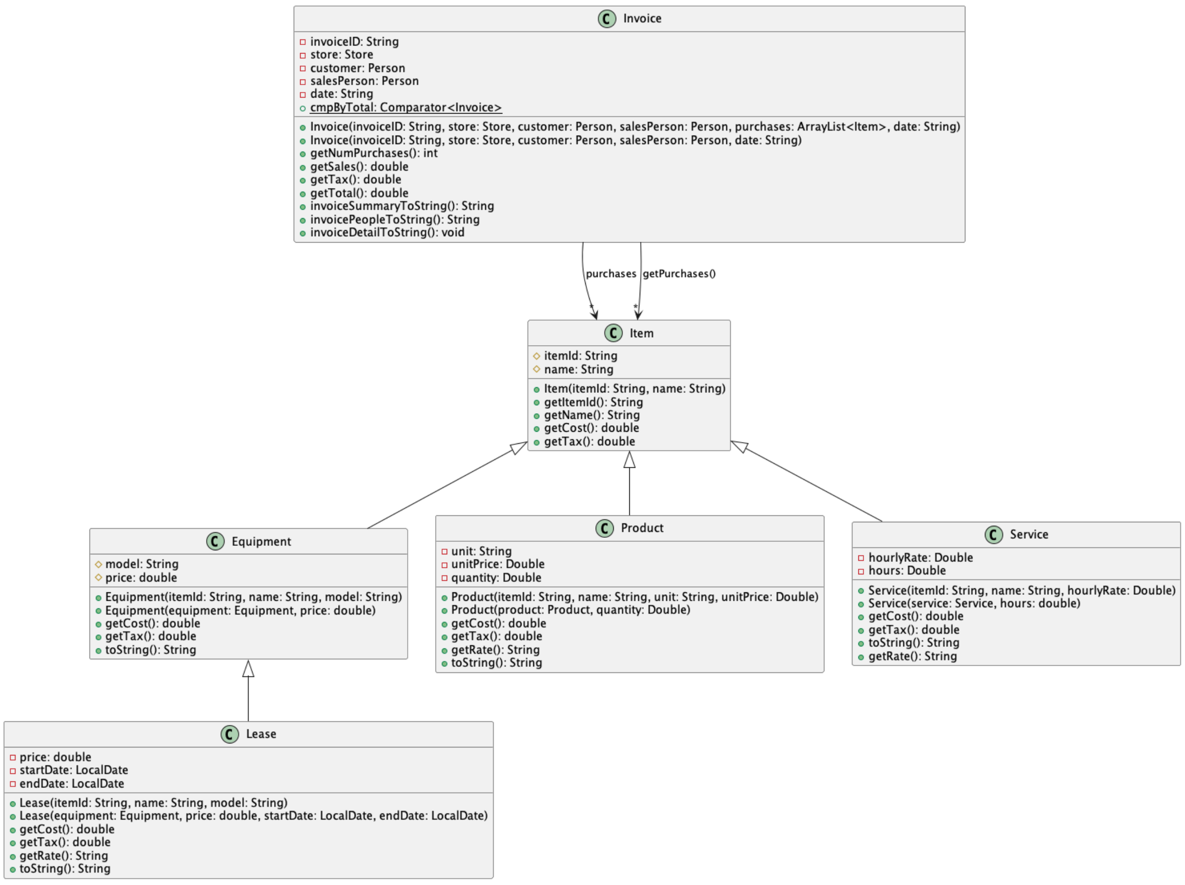

Invoice
invoiceID: String
store: Store
customer: Person
salesPerson: Person
date: String
cmpByTotal: Comparator<Invoice>
Invoice(invoiceID: String, store: Store, customer: Person, salesPerson: Person, purchases: ArrayList<Item>, date: String)
Invoice(invoiceID: String, store: Store, customer: Person, salesPerson: Person, date: String)
getNumPurchases(): int
getSales(): double
getTax(): double
getTotal(): double
invoiceSummaryToString(): String
invoicePeopleToString(): String
invoiceDetailToString(): void
purchases
getPurchases()
*
*
Item
itemId: String
name: String
Item(itemId: String, name: String)
getItemId(): String
getName(): String
getCost(): double
getTax(): double
Equipment
model: String
price: double
Equipment(itemId: String, name: String, model: String)
Equipment(equipment: Equipment, price: double)
getCost(): double
getTax(): double
toString(): String
Product
unit: String
unitPrice: Double
quantity: Double
Product(itemId: String, name: String, unit: String, unitPrice: Double)
Product(product: Product, quantity: Double)
getCost(): double
getTax(): double
getRate(): String
toString(): String
Service
hourlyRate: Double
hours: Double
Service(itemId: String, name: String, hourlyRate: Double)
Service(service: Service, hours: double)
getCost(): double
getTax(): double
toString(): String
getRate(): String
Lease
price: double
startDate: LocalDate
endDate: LocalDate
Lease(itemId: String, name: String, model: String)
Lease(equipment: Equipment, price: double, startDate: LocalDate, endDate: LocalDate)
getCost(): double
getTax(): double
getRate(): String
toString(): String

Figure 3: UML diagram for Items

the instructions laid out by FMT. The data is parsed and then instances of people, stores, addresses, items (equipment, products, and services), invoices, and invoice items (purchased equipment, leased equipment, purchased products, and commissioned services) are created. The data needs to be outputted to valid JSON[1] or XML[3]. The data loading is tested in various steps:

  - Ensure the CSV loader is parsing at commas by outputting parsed data and comparing to the CSV file.
  - Ensure the person loader creates accurate instances of people. The primary pitfall may occur when creating in instances of people without emails or with long lists of emails. Data including both types of people needs to be included.
  - Ensure the item loader creates accurate instances of items. Items can be equipment, products, or services. Testing needs to include data for each type of item.
  - Ensure the store loader creates accurate instances of stores. Each store has a manager. Testing must be done to ensure the store loader is correctly interacting with the people that have been loaded.
  - Ensure the purchase loader creates an accurate mapping of invoices to purchases.
  - Ensure the invoice loader correctly creates invoice instances. It is possible that an invoice has no purchases listed so that case needs to be tested for in the data. The invoice interacts with all the other data loaded.

Three invoices are tested with items of each type. An equipment item is both purchased and leased to ensure that tax is not being applied to the purchases. Equipment of different values are leased to ensure the taxes scales with the cost correctly. Many items are listed for a different number of days to ensure the prorated cost is applied correctly. Products with many units are tested and tax rate is added and services are tested similarly.

- **Report Generation** The data that is loaded into instances of basic Java objects need to be printed to XML and JSON files. FarMarT has also requested various sales reports:

  1. A report to give a summary of all sales along with a few totals.
  2. A report to give a similar summary but for each store.
  3. A report to give details for each individual invoice.

  The serialized data needs to be in valid XML and JSON format which is verified using the following vaidator: https://www.w3schools.com/xml/xml_validator.asp. Furthermore, the sales reports for the test invoices described above are computed by hand to test that the reports are generating the correct values.

3.3 Database Interface

The Java application described above connects to a MySQL database using JDBC. Records are loaded from the database and create instances of Java objects by using a handful factory methods. Furthermore, data can be persisted, modified, and deleted using the application. The methods are described below:

- **Items** The database is queried to retrieve instances of (generic, unpurchased) items with a specified item type. Each item type has different data associated to it and so there is a method associated to each item type.
- **People** All people are retrieved from the data base.
- **Stores** All stores are retrieved from the database and the manager is added to the store since people have already been loaded.
- **Invoice Items** Purchased items are retrieved from the data base and added to an invoice with the salesperson and customer and then added to the appropriate store's list of invoices.

All persisting of data first checks to see if a record already exists.

3.3.1 Component Testing Strategy

Testing for this component is analogous to testing for the class/entity structure. However, additional testing is implemented to ensure data persisting is functioning well. Some test data is persisted using the API and the database can be tested using the queries described above.

3.4 Design & Integration of Data Structures

An ordered list data structure is implemented to print reports according to client specifications. The ordered list takes as input a comparator to determine how the list will be ordered. The following comparators are implemented:

- compare an invoice by customer last name then first name in alphabetical order.
- compare an invoice by the total sale amount from highest to lowest.
- compare an invoice first by store number then by sales person name.

3.4.1 Component Testing Strategy

Test cases are implemented for each comparator. A list of invoices where customers have the same last name but different first names is used to ensure the comparator is functioning correctly. A similar test is used for sales person names in the third comparator.

3.5 Changes & Refactoring

During development an Invoice Item class was created. This broke encapsulation principles and so the design was refactored to remove this superfluous class.

References

[1] GSON Library. https://github.com/google/gson. [Retrieved March 02, 2023].

[2] Log4j Library. https://logging.apache.org/log4j/2.x/. [Retrieved April 20, 2023].

[3] XML Library. http://x-stream.github.io/. [Retrieved March 02, 2023].