VGB's Modernization

Design Document

Henry Monahan
hmonahan2@huskers.unl.edu
Chi Hieu Phan
cphan5@unl.edu
University of Nebraska—Lincoln

Spring 2025
Version 7.0

This report provides a detailed overview of the functionality, design, and implementation of a program designed to modernize Ron Swanson's company VGB.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Henry Monahan, Chi Hieu Phan	2025/2/14
2.0	Adding class functionality, polishing earlier changes	Henry Monahan, Chi Hieu Phan	2025/2/28
3.0	Added a report functionality	Henry Monahan, Chi Hieu Phan	2025/3/16
4.0	Finalized ER diagrams, testing strategies, and query examples	Henry Monahan, Chi Hieu Phan	2025/4/2
5.0	Added database schema, JDBC API connection established	Henry Monahan, Chi Hieu Phan	2025/4/15
6.0	Added data manipulation and persisting details	Henry Monahan, Chi Hieu Phan	2025/4/22

Table of Contents

1. Introduction

Ron Swanson's company, Very Good Building & Development Company (VGB) is expanding and needs to modernize. Currently, the company keeps track of all of its data in spreadsheets and physical paper. VGB's business is involved in all aspects of construction: they act as a general contractor creating and awarding subcontracts, sell, lease, and rent construction equipment and building materials related to construction. Swanson wants to update all business operations with newly developed systems for inventory, marketing, delivery, invoicing, and sales. This program registers, sorts, and organizes the data, making for more efficient keeping of inventory.

Client & Business Model

The client is a construction/equipment rental company that:

- Manages inventory of equipment (purchases, leases, rentals), materials, and subcontractor contracts
- Tracks customers (companies) and their contacts (persons)
- Records sales transactions with salespersons
- Needs to generate comprehensive invoice reports
- VGB requires a MySQL database to replace flat-file storage, ensuring ACID compliance for invoice transactions and supporting concurrent access

Key business rules:

1. Different tax rates apply based on transaction type and amount
2. Equipment can be:
    - Purchased: 5.25% tax
    - Leased: 50% Markup when ametorized over 5 years. If more than $12500, a flat tax of $1500 would be applied
    - Rented: 0.1% of retail price per hour with a 4.38% tax
3. Materials are sold by unit with quantity-based pricing, taxed at a rate of 7.15%
4. Contracts have fixed amounts with subcontractors, with no tax involved
5. Invoices may contain multiple items of different types

1.1 Purpose of this document

The purpose of this document is to document the subsystem responsible for keeping track of all invoices and billing, as well as developing reports of the processed data in a database-backed system.

1.2 Scope of the Project

This project creates, organizes, and reports Invoice data. Out of the objectives in Ron Swanson's grand vision: inventory, marketing, delivery, invoicing, and sales- this document only tackles invoices, leaving the rest to other teams. Covered in this document are design choices about class structure, database connectivity, and the data invoice reports that are generated. There will be no methods/uses for handling the inventory, marketing, or delivery aspects of VGB.

This project does cover and uses several key components. First, CSV to XML/JSON conversion entails transforming data from CSV format into XML or JSON formats for better data interchange. Second, the class hierarchy and JUnit testing phase involves designing a structured class hierarchy and implementing unit tests to ensure code reliability.
Third, invoice reporting focuses on generating detailed reports from invoice data.
Fourth, database design involves structuring the database to efficiently store and manage data. Fifth, database connectivity ensures that the application can interact with the database seamlessly. Lastly, using an API to interact with the database allows for efficient data manipulation and retrieval through defined endpoints.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Class hierarchy: A class hierarchy (also known as an inheritance hierarchy) is a way of organizing classes in object-oriented programming where classes inherit properties and behaviors from parent classes, creating a tree-like structure of relationships.

Database interface: A database interface in Java provides an abstraction layer between application code and database operations

Map: A type of list in which two types of values are linked together in which case one type can be fetched with the other (i.e. Person, PersonId)

Primary Key: A key for database searching and sorting

1.3.2 Abbreviations & Acronyms

ACID: "Atomicity, Consistency, Isolation, Durability—transaction guarantees enforced by MySQL."
API: Application Programming Interface
EDI: Data Representation & Electronic Data Interchange
IDE: Integrated Development Environment

JDBC: "Java Database Connectivity API for SQL database interaction."
Junit: A Java Testing Program

OOP: Object Oriented Programming
Single-Table Inheritance: "Database strategy storing subtypes
UUID: Universally Unique ID
VGB: Very Good Building and Development Company

2. Overall Design Description

This system uses a class-based data structure to represent the various people, companies, and items involved with the billing and invoices. Classes are designed for each of these objects, with subclasses representing details as necessary.

The overall architecture of the system follows a clean, modular design grounded in object-oriented programming (OOP) principles. The data flow begins with CSV files that are parsed into Java objects using custom parsers. These objects represent core business entities such as persons, companies, equipment, materials, and contracts. Once loaded, the objects are organized, manipulated, and summarized using calculators and report generators. Reports are then produced in a clear, tabular console format, mimicking invoice summaries and breakdowns. During the early phases, Java was selected as the primary language due to its strong OOP capabilities, type safety, and robust libraries. JUnit was used for effective unit testing of class logic. As the project matured, the application transitioned from flat-file data handling to database-backed interaction using JDBC. Data now flows from a remote MySQL database through JDBC into the Java application, where ResultSets are converted into corresponding objects. This design ensures separation of concerns, where data access, business logic, and reporting remain cleanly divided. To enhance report capabilities and maintain ordered collections of data, a custom generic `SortedList` ADT was implemented. This node-based data structure accepts a Comparator to maintain internal ordering without relying on external sorting methods or built-in collections. It is now used in key reporting features such as listing invoices by total, by customer name, and summarizing total invoices per company. Together, these technologies—CSV parsing, JDBC connectivity, custom data structures, and OOP in Java—combine to form a robust and maintainable system for VGB's invoicing needs.

2.1 Alternative Design Options

Every piece of data is loaded from the database and mapped to the Primary Key that is loaded along with it. After it is loaded in, all managing of objects is done inside the IDE using the loaded Primary keys to relate objects.

An alternate design option is every object isn't loaded at once, but instead loaded as they were needed. For example, if an invoice has a foreign Person ID of 5, the database would find the person who had a Primary Key of 5 and then and only then load that person,

turning it into an object. This method wasn't used because it's overly complicated and slower, as the connection to the database would have to be made laboriously repeatedly.

3. Detailed Component Description

3.1 Database Design

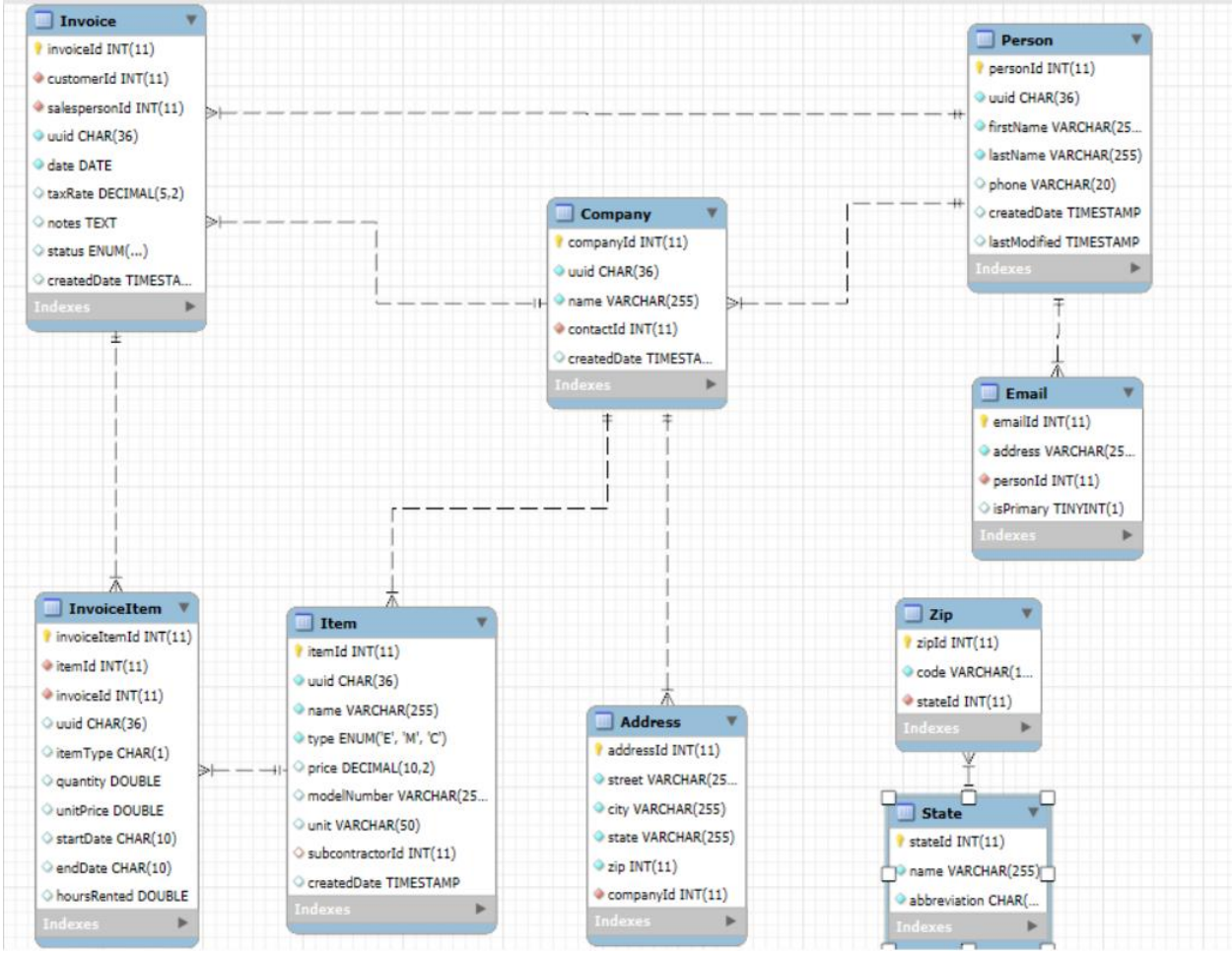

Figure 1: ER diagram of the table structure

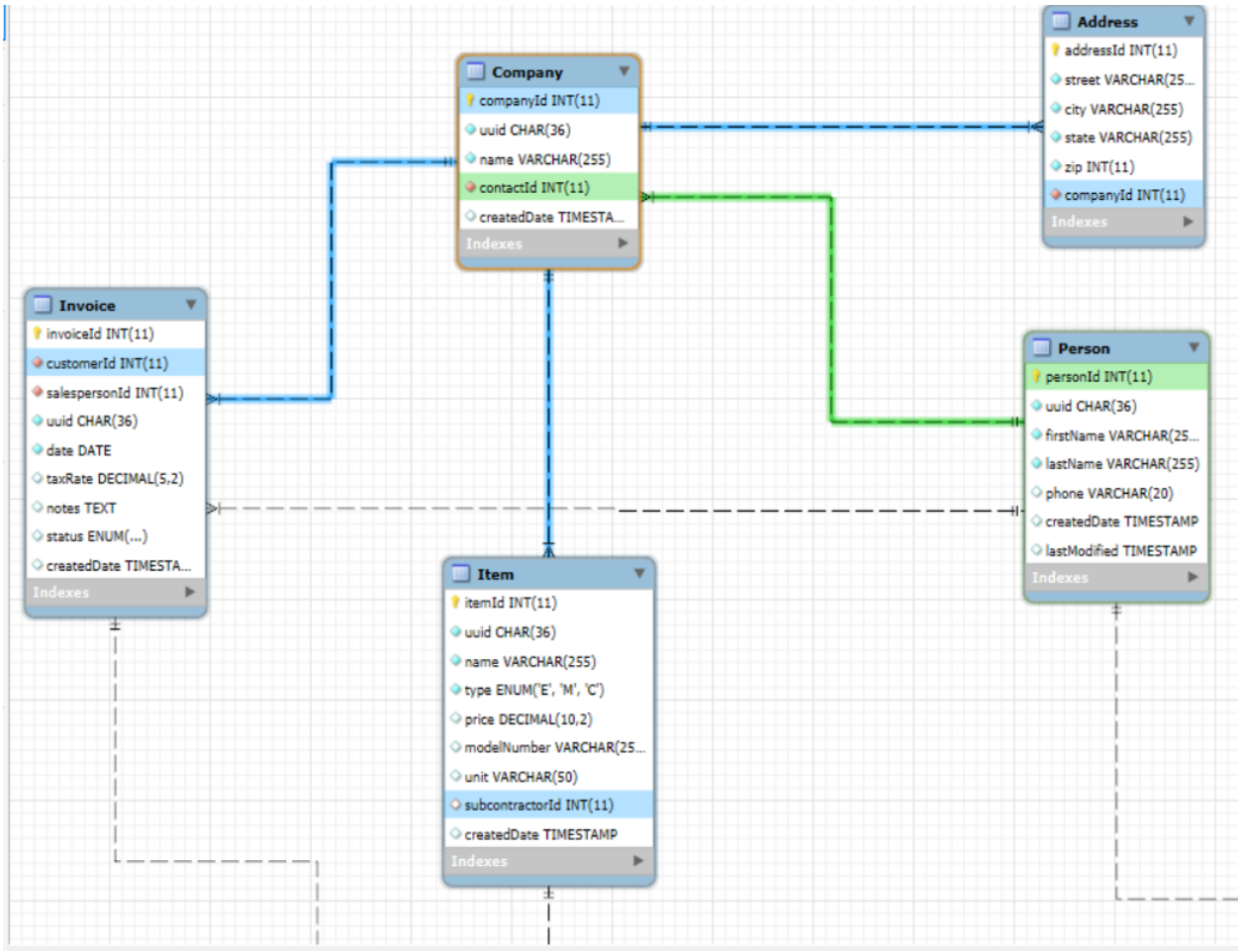

Figure 2: contactId from Company is linked to PersonId.

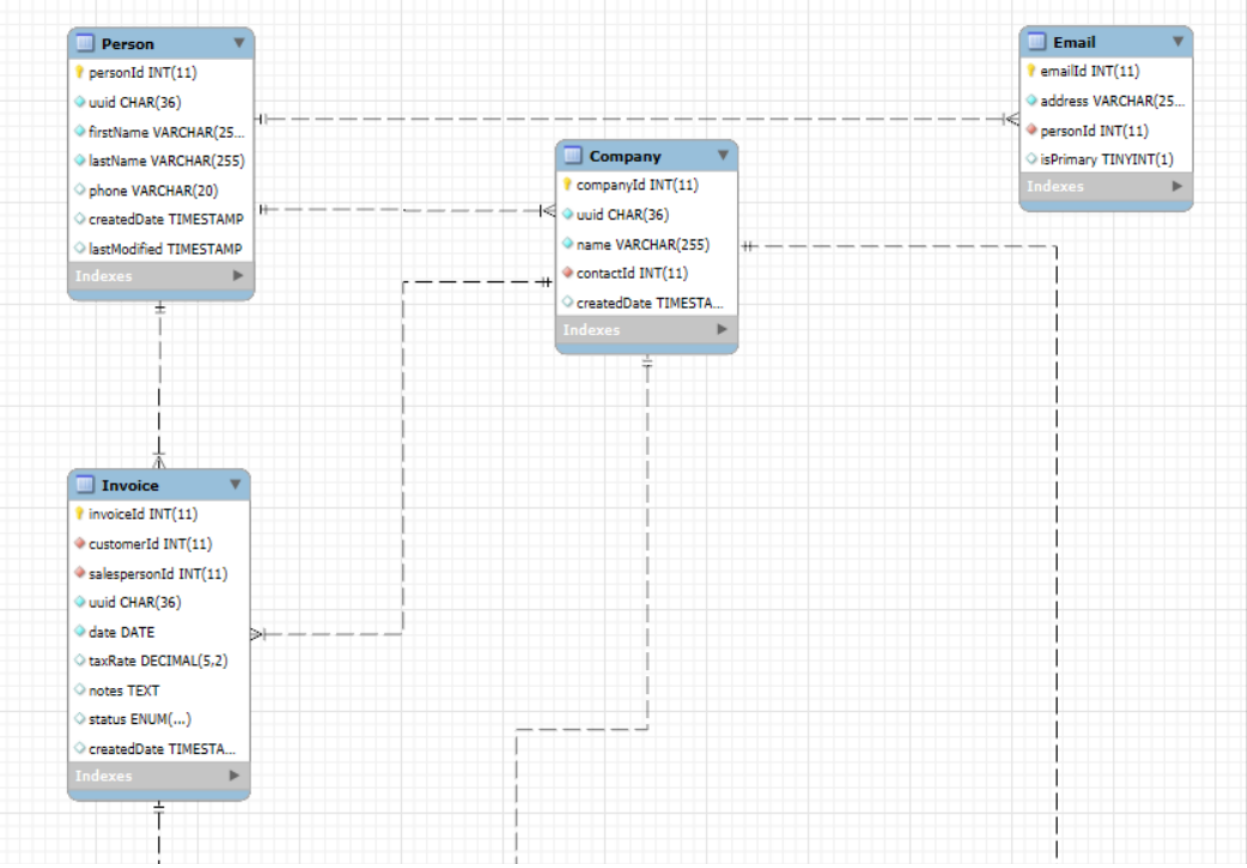

Figure 3: personId from Person is linked to personId in Email, ContactId in Company and salespersonId in Invoice(show the saleperson)..

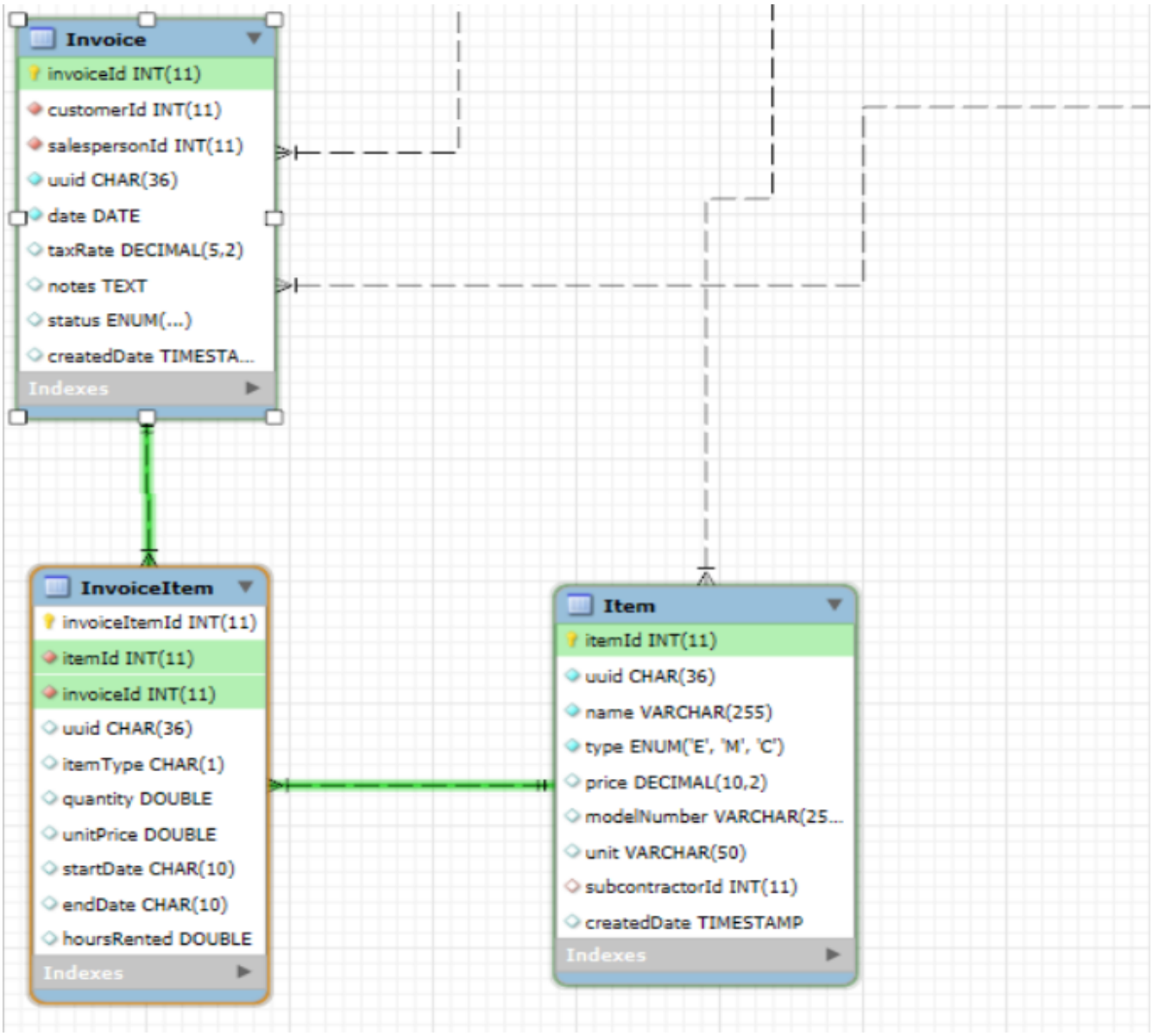

Figure 4: itemId from InvoiceItem is linked to itemId in Item. invoiceId from InvoiceItem is linked to invoiceId in Invoice.

3.1.1 Component Testing Strategy

Tools & Methodology

- MySQL 8.0 on nuros.unl.edu
- Queries were manually run and verified using SQL script
- Data was generated via inserts and validated against manual calculations
- Edge cases were tested explicitly

Breakdown of Queries and Purpose

Query No.	Purpose	Test Description	Result
1	Retrieve UUID and names of all persons	Checked correct parsing and joins of Person table	Pass
2	Retrieve detailed person info with their emails	Tested multi-row join from Person to Email using foreign keys	Pass
3	Get emails of a specific person	Parameterized test with a known UUID	Pass
4	Update email of a specific record	Verified update took effect with SELECT	Pass
5	Remove a person record	Tested foreign key restrictions and ON DELETE behavior	Pass
6	Get all items in a specific invoice	Tested invoice-UUID join across invoice_items and item tables	Pass
7	Retrieve items purchased by a specific customer	Filtered through company UUID; tested join on Invoice + InvoiceItem	Pass

8	Count of invoices per customer (including zero)	LEFT JOIN and GROUP BY on Invoice and Company	Pass
9	Count of invoices per salesperson (only those with sales)	INNER JOIN to exclude 0-sale cases	Pass
10	Subtotal cost of materials in each invoice	Aggregated quantity × cost without tax; used GROUP BY	Pass
11	Detect duplicate material entries in a single invoice	Used COUNT(*) + GROUP BY having clause to expose duplicate entries	Detected
12	Detect potential fraud (same person as contact and salesperson)	Used join condition comparing contact UUID to salesperson UUID	Detected

Key Observations & Fixes

• Duplicate Materials Detected: Certain invoices included repeated material entries. This was resolved by aggregating duplicates in the application logic before database insertion.

• Fraud Detection Effective: Edge cases were inserted with overlapping contact and salesperson UUIDs, and the fraud query successfully flagged them.

3.2 Class/Entity Model

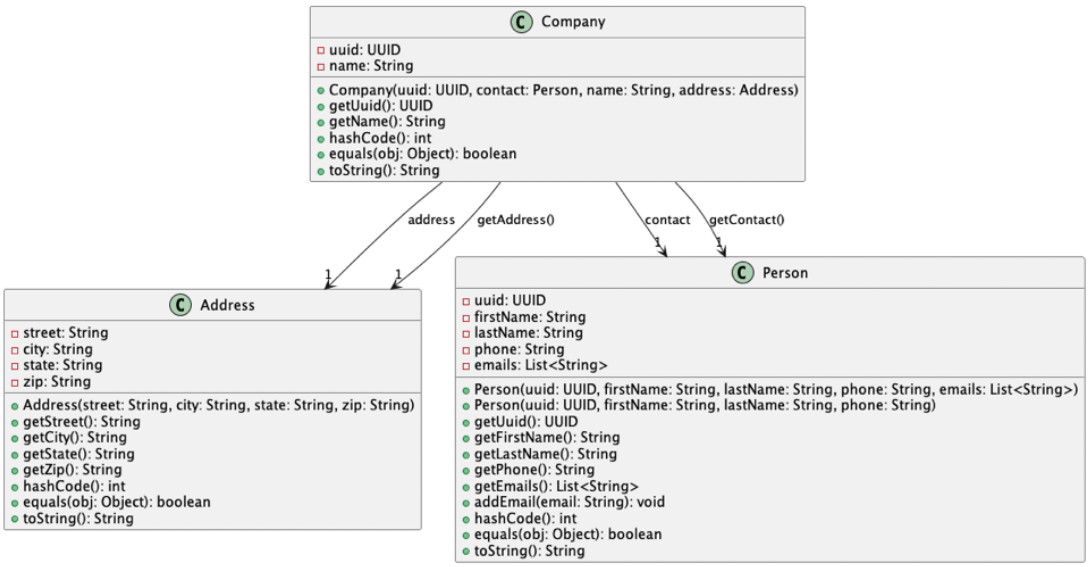

Figure 1: This model represents the basic relationships between the Company, Person, and Address classes as are necessary in the designed data structure

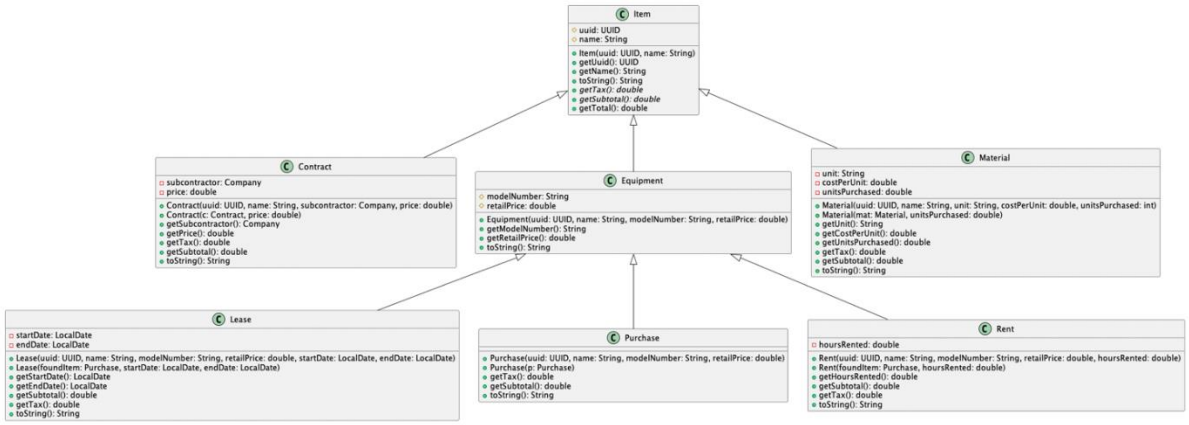

Figure 2: This shows the Item inheritance structure. Both Item and Equipment are abstract classes and have functions only to be implemented by the Lease, Purchase, and Rent classes

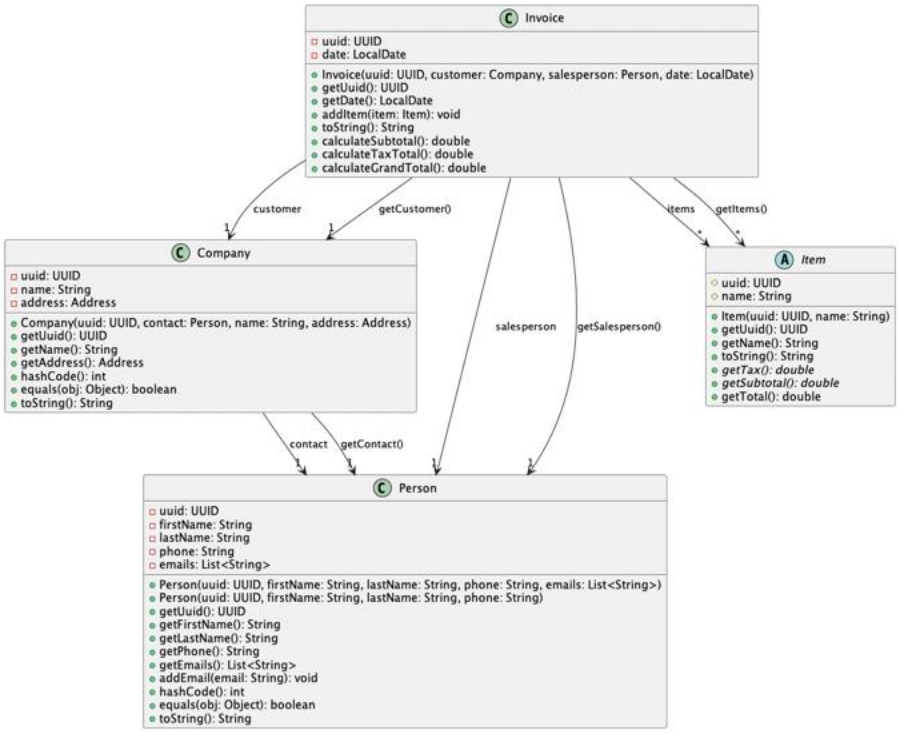

Figure 3: Relationship between Invoice and the classes it uses

This program will use Object Oriented Programming in Java as well as a Database in MySQL to accomplish these objectives. The classes that will be used are as follows:

Class: Person

Fields: UUID, Name, Email Address

Use: Represents people involved in the business, often mapped to UUIDs to allow consistency when navigating other classes.

Class: Company

Fields: UUID, Contact UUID (Primary Contact for the Company), Name, Address

Use: Data that represents a company is read into a list and stored as Company class variables.

ABSTRACT Class: Item

Fields: UUID, Name, Type

Use: Data pertaining to each available item (equipment, material, contract) is read into a list and stored as Item class variables as a superclass and depending on the type is stored in either an Equipment, Material, or contract subclass. The getCost() method is an abstract method implemented/overridden by the subclass.

ABSTRACT Class: Equipment (Extends Item)

Fields: Model Number, Retail Price

Use: Secondary step on the Item class- does not implement any methods and leaves them to its subclasses

Class: Purchase (Extends Equipment)

Fields: N/A

Use: Implements getTax() and getSubtotal() from the Item class as an up-front purchase.

Class: Lease (Extends Equipment)

Fields: Start Date, End Date

Use: Implements getTax() and getSubtotal() from the Item class as a lease, this time taking the amount of time leased calculated from Start and End dates read in through the invoice items file.

Class: Rent (Extends Equipment)

Fields: Hours Rented

Use: Implements getTax() and getSubtotal() from the Item class as a Rental purchase-calculates using the amount of hours rented

Class: Material (Extends Item)

Fields: Unit, Cost per Unit, Units Purchased

Use: Bottom tier specification of equipment data. Implements “getCost()”.

Class: Contract (Extends Item)

Fields: UUID corresponding to Subcontract company (as specified in companies data file)

Use: Bottom tier specification of equipment data. Implements “getCost()”, this time not adding taxes on the end of the calculation.

Class: Invoice

Fields: Invoice UUID, Customer UUID, Salesperson UUID, and Date

Use: Represents an Invoice item and connects to almost every other class through a foreign key.

Class: Address

Fields: Street, City, State, and Zip

Use: Represents an address of a company’s headquarters. In this

3.2.1 Component Testing Strategy

Tools Used: JUnit 5

Approach: Employed black-box and white-box testing for entity constructors, accessors, and financial calculations including getSubtotal(), getTax(), and getTotal() implementations.

Class Tested	Test Type	Number of Tests	Test Description
Purchase	Unit Test	3	Test 5.25% tax rule across boundary price points.
Lease	Unit Test	3	Test lease amortization and flat tax application.
Rent	Unit Test	2	Verify hourly rental rate and rounding.
Material	Unit Test	2	Test unit price x quantity with tax calculation.
Contract	Unit Test	2	Test fixed contract amount and tax exclusion.
SortedList	Functional Test	3	Verify ordering, insertion, and deletion behavior.

3.3 Database Interface

MySQL Database Integration with Java (JDBC)

The integration of the MySQL database with the Java application was achieved through the use of Java Database Connectivity (JDBC). The following steps were followed to establish a seamless connection:

- Driver Registration: The MySQL JDBC driver was made available to the project to enable communication with the database.
- Database Connection Establishment: A utility method (getConnection) was created within the application to establish a connection to the database. This method configured the database URL, username, and password, and leveraged the DriverManager class to open the connection.
- Resource Management: All database interactions were encapsulated within Java's try-with-resources blocks to ensure that connections, statements, and result sets were automatically closed after operations, preventing resource leaks.
- Error Handling: Database operations were wrapped in exception handling mechanisms to catch and properly respond to any SQLException that might occur during runtime.

Retrieving and Mapping Database Data to Java Objects

The retrieval and mapping of data from the database to Java objects were organized systematically:

- Data Retrieval: SQL SELECT queries were executed against the database to fetch records from the Person, Company, Item, and Invoice tables.
- Result Processing: Each query returned a ResultSet, which was iterated over to extract individual field values corresponding to columns in the database.
- Object Mapping: The extracted data was used to instantiate Java objects such as Person, Company, Material, Contract, Invoice, etc.
- Each object accurately reflected a row in the database and held data in its corresponding fields.
- Object Storage: After creation, the objects were stored in Java collections (e.g., HashMap<Integer, Object>) to allow efficient retrieval, association, and further processing within the application.
- Foreign Key Resolution: Special care was taken to resolve foreign key relationships. For example, a Company object references a Person as its contact, and an Invoice references both a Company (customer) and a Person (salesperson).

Saving Java Objects Back to the Database

The process of persisting Java object data back into the MySQL database followed a structured workflow:

- SQL Statement Preparation: INSERT statements were formulated with parameter placeholders to accept Java object data securely and efficiently.

- Binding Data: Before executing a statement, the attributes from Java objects (such as UUIDs, names, prices, addresses) were bound to the appropriate placeholders.

- Executing Updates: The database was updated by executing the prepared statements, thereby inserting new records or modifying existing ones based on the Java objects' state.

- Maintaining Referential Integrity: While inserting, attention was given to foreign key constraints, ensuring that referenced entities already existed in their respective tables.

- Transaction Considerations: Although not explicitly batched or transactionally grouped in this phase, the operations were atomic at the individual statement level.

3.3.1 Component Testing Strategy

Approach- a combination of self- produced white box testing and third party black box testing. This part of testing was to ensure JDBC was functional, so we tested one of each

type of SQL defined table- other than Item, for which we tested 5- one for both Material and Contract, and 3 for Equipment (Lease, Rent, Purchase). In addition for each of these, we tested data that didn't fit the database's requirements, such as adding feeding a null value into a field that had a "not null" specifier.

Table Tested	Test Description	Number of tests
Item	Test 5.25% tax rule across boundary price points.	1
Lease	Test lease amortization and flat tax application.	1
Rent	Verify hourly rental rate and rounding.	1
Material	Test unit price x quantity with tax calculation.	1
Contract	Test fixed contract amount and tax exclusion.	1
SortedList	Verify ordering, insertion, and deletion behavior.	1
Person	Verify Person is loaded with emails	1
Company	Verify company is loaded, each with connection to person through foreign Person key	1
Address	Verify connection to company	1
Invoice Item	Test connection to the fields it inhibits- Person, Company, Invoice	1
Invoice	Test cross-relation with Person, Company, and InvoiceItem	1

3.4 Design & Integration of a Sorted List Data Structure

The custom data structure implemented was node-based. The custom structure was developed with a clean interface that defines standard list operations, including:

- add(Element e): Insert a new element while maintaining sorted order.
- remove(Element e): Remove an element from the list.
- contains(Element e): Check if an element exists.
- size(): Return the number of elements.
- get(int index): Retrieve an element at a specific position.

The custom data structure is designed to be generic because: To allow the same list structure to hold and manage any type of object, not just a specific class; promotes reusability and type safety, allowing compile-time checks and avoiding casting errors; enables the structure to adapt easily to new data types as the project expands without rewriting or duplicating the list logic.

Sorted data is expected to need in following order:

- A list of all invoices ordered by the total highest-to-lowest
- A list of all invoices ordered by the total customer (name) in alphabetic ordering
- A list of all customers (companies) ordered by the total of all their invoices.

3.4.1 Component Testing Strategy

Tools & Objectives

Tools Used: Java (manual `System.out.println` and assertions during development)

Objectives:

- Ensure correct insertion order using the provided comparator
- Validate remove() handles head, middle, and tail cases
- Confirm get(index) behaves correctly
- Verify size tracking is accurate
- Test iteration through the list using Iterable interface

Test Cases

Test Case	Input/Action	Expected Behavior	Result
Add to empty list	Add 1 item	Head points to item, size = 1	Pass
Add in order	Add A < B < C	Order maintained: A → B → C	Pass
Add out of order	Add C, A, B	List becomes A → B → C	Pass
Add duplicates	Add A twice	Both entries appear in order	Pass
Remove head	Remove A from A → B → C	Head updated, size decreases	Pass
Remove tail	Remove C from A → B → C	Tail removed, list updates	Pass
Remove middle	Remove B from A → B → C	List becomes A → C	Pass
Get valid index	Get(1) from A → B → C	Returns B	Pass
Get invalid index	Get(-1) or Get(5)	Throws IndexOutOfBounds	Pass
Size validation	After adds/removes	Reflects real count	Pass
Iterator coverage	Use for-each loop	Iterates all items in order	Pass

Observations

• Comparator-based ordering worked well for numeric and custom object types.

• Edge cases like empty list removal or invalid index access were safely handled via exceptions.

• The Iterator allowed smooth integration into enhanced for-loops, aiding in report generation.

Bibliography

[1] *APA 6 – Citing Online Sources*. (n.d.). Retrieved March 19, 2021, from https://media.easybib.com/guides/easybib-apa-web.pdf

[2] Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

[3] MySQL 8.0 Reference Manual. (2023). Oracle.

[4] JDBC Tutorial. (2023). Oracle. https://docs.oracle.com/javase/tutorial/jdbc/

[5] What is many-to-many relationship https://vertabelo.com/blog/many-to-many-relationship/