Very Good Building Invoice Subsystem

Made for the Very Good Building & Development Company

Gavin Blesh
gblesh2@huskers.unl.edu
University of Nebraska—Lincoln

Spring 2025
Version 6.0

This document covers the development behind the Very Good Building company's Java-based application. It is built upon Object-oriented Java principles, a Structured Query Language (SQL) database, and Sorted List data types.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Gavin Blesh	2025/14/02
2.0	Updated draft that incorporates class hierarchy.	Gavin Blesh	2025/28/02
3.0	Updated draft that incorporates an SQL database design	Gavin Blesh	2025/28/03
4.0	Updated draft that incorporates a database interface.	Gavin Blesh	2025/11/04
5.0	Updated draft that incorporates a Sorted List Data Structure	Gavin Blesh	2025/23/04
6.0	Updated draft that applies final changes and cleans up the document. Serves as the final draft of the document	Gavin Blesh	2025/05/09

Table of Contents

1. Introduction

The following project is a large-scale database-backed application built in Java using Object-Oriented programming and an SQL database. It is built for the Very Good Building & Development Company(VGB). VGB is a general contractor in the construction industry that handles subcontracts, leases, sells, and rents construction equipment and building materials. VGB wants to modernize the company and move all its data from physical copies to digital copies. The company's CEO, Ron Swanson, wants independently developed systems for inventory, marketing, delivery, invoicing, and sales.

The purpose of this application is to develop an invoice subsystem that keeps track of all invoices and billing in a database-backed system. The invoice subsystem is designed in java using Object-Oriented design principles to help support VGB's business model and rules while also providing their desired functionality.

Taxes are assessed to an invoice based on the type of item on an invoice. Materials on an invoice have a 7.15% sales tax. On the other hand, Contracts on an invoice are not charged a sales tax. Equipment is charged a sales tax, but how much depends on how the equipment is charged to the customer. Equipment can either be purchased, rented, or leased. Purchased equipment requires the buyer to pay in full and is charged a 5.25% use tax. Rented equipment, which charges the customer .1% of the retail price for each hour rented, is imposed a 4.38% sales tax. Leased equipment is equipment that is signed over for use for a period of time and its cost is determined by the cost of the lease amortizing the retail price over 5 years, prorated to the lease period, with a 50% markup included. How much tax is applied to a lease is determined by the equipment's retail price. If the retail price is under $5000, no tax is applied, if the retail price is between $5000 and $12,499.99, then the tax applied is $500. Anything that is $12,500 or more is charged a tax of $1500.

The database backed application design also has the Java application read in data from the SQL database and output summary reports for the invoices.

1.1 Purpose of this Document

The purpose of this design document is to layout and design the Very Good Building Invoice Subsystem program so that VGB is able to understand the program's capabilities and how it functions. This document describes the program's overall design and goes further in depth to describe it including the class/entity model design, the database design, the database interface, and how each of those sections are tested. It also covers the design and incorporation of a sorted list data structure.

1.2 Scope of the Project

- This project creates a database for VGB to store its data online instead of physically on paper.
- Each system of VGB's project is developed independently of the others. This project only covers the invoicing system and does not cover the systems for inventory, marketing, delivery, and sales, or the user interface. The purpose of the invoicing

system is to break down VGB's invoices into different parts and to create summary reports.

  - Three summary reports are generated by the application:
    - 1. A summary report that outputs the total of the invoices
    - 2. A summary report that outputs the total invoices for all customers. This includes the total of all of a customer's invoices added up.
    - 3. A summary report of each individual invoice
  - A Linked List data structure is used to generate another set of three summary reports:
    - 1. A summary report that sorts the invoices by total
    - 2. A summary report that sorts the invoices by company name
    - 3. A summary report that sorts the customers by the total of all their invoices.

- Each VGB invoice includes a Universally Unique Id (UUID), a date, and each item on the invoice.
  - When the item is equipment, it is broken down into rent, leased, or purchased, with each having its own additional data that determines how much the customer pays and when.
  - If the item is considered material, it is broken down into units sold, with each unit having a particular price per unit.
  - Furthermore, if the invoice includes contracts, the contract and invoice will include the subcontractor company information and their total amount.
- Additional taxes paid are also present on the invoice.
- The project also uses class/entity models and a database interface to design VGB's database. It implements certain functionalities such as data/file loading; class hierarchies, which allow for a more robust system that is less prone to errors; and testing strategies that ensure the program functions properly.
- A sorted list data structure is also used and integrated into this application to help design the database.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Contract Category of items that are negotiated and managed by VGB for customers, which are serviced by subcontractors. Each contains a customer/company, contractor, and total amount charged. No tax is applied to contracts.

Customer A company that is assigned to an invoice

Equipment Category of items that includes heavy equipment (i.e., bulldozers) and small equipment (i.e., jackhammers). Each contains an alphanumeric model number and retail price. Companies are given the option of purchasing, leasing, or renting Equipment, which determines the tax.

Lease A method of buying Equipment that involves having equipment signed over to the customer for a period of time. Its cost is determined by the cost of the lease amortizing the retail price over 5 years, prorated to the lease period, with a 50% markup included. If the retail cost is under $5000, no tax is applied. If the retail cost is between $5000 and $12,499.99, a $500 tax is applied. Lastly, if the retail cost is $12,500 or more, a tax of $1500 is charged.

Material Category of items that includes items such as items, nails, rebar, etc. Sold in quantities by unit (i.e., nails being sold in 10lb boxes or lumber being sold in linear feet). Each includes a unit and retail price and faces a 7.15% sales tax.

Purchase A method of buying Equipment that involves the customer paying the full retail price. All equipment purchases have a 5.25% use tax applied.

Rental A method of buying Equipment that involves the customer signing a lease to borrow equipment for a specified time period, where they are charged .1% of the retail price each hour rented. A 4.38% sales tax is applied to all rentals.

1.3.2 Abbreviations & Acronyms

API Application Programming Interface

CEO Chief Executive Officer

CSV Comma Separated Values

JDBC Java Database Connectivity API

JSON JavaScript Object Notation

SQL Structured Query Language

UML Unified Modeling Language

UUID Universally Unique ID

VGB Very Good Building & Development Company

2. Overall Design Description

The application's invoicing system keeps track of every invoice and every item that appears on an invoice. Each invoice includes a Universally Unique Identifier (UUID), a creation date, the customer's name, address, and primary contact, and all items that appear on the invoice. Any taxes on items are also applied in the invoice. Additionally, each item, whether it is considered equipment, materials, or contracts have their own additional data the program accounts for, which is covered in 1.3.1 of this document, which is the definitions section.

Furthermore, this database application can manage and store invoice data, calculate/generate the data needed for invoice reports, and load that data to produce reports. Key components of the application include:

- Classes that represent entities included in the invoice. Additionally, every class in this application is built to be stand-alone so that it can be integrated into different projects and is not reliant on anything else in the application.
- The ability to output the data into formatted JavaScript Object Notation (JSON) files.
- The ability to read invoice data from flat files and produce different reports based on that data. This is accomplished in a similar manner to the class entities mentioned earlier. Stand-alone classes are used to read in the invoice data and the program then computes what is needed before outputting them.
- The function of generating three different reports based on the data. The first gives a summary of all invoices. The second report gives a summary for each customer. Finally, the third report gives details on individual invoices is the third report.
- A relational database designed in SQL that supports, models, and stores the data and objects involved in the Java application. SQL tables are designed in the database to store the data and objects and any data related to them. Furthermore, the database models the relationship between the data using foreign keys.
- The ability to load in objects from the SQL database into the Java application and produce different reports based on that data. This is accomplished via an API that connects the database to the application.
- The ability to manipulate data in the Java application and use an API to reflect these changes in the SQL database. This is accomplished via a Java Database Connectivity API (JDBC)
- The ability to store objects in a sorted list data structure and generate reports. This data structure holds an arbitrary number of objects in it, sorts them, and orders them, with any tie breakers being done by comparing the UUID numbers of the objects being sorted.

  - This sorted list data structure generates three more additional reports, which sort the invoices by total amount, the invoices by customer name, and the customers by the total of all their invoices

2.1 Alternative Design Options

An alternative design considered for the application included having a singular class that loaded in all three inputs and a singular class that outputs all three to JSON files. The reason that this was not implemented was because it breaks the single responsibility principle and makes it so that the program and its classes are not stand-alone. The current design was chosen because:

- It allows all aspects of the program to act on a stand-alone basis and not tie into each other.
- It does not violate the single responsibility principle of coding.
- The alternative approach could lead to breaking encapsulation as well in other parts of the application.
- The ability to connect the classes via class hierarchy makes it so the program is less prone to errors in other areas of the project. In this case, the advantages of incorporating class hierarchy outweigh the disadvantages.

Another design considered for the application included having all the data from the classes printed in GenerateReports.java. However, this design would break encapsulation, so the print functions were instead put in the object's respective classes and called through toString() methods. The latter design was chosen because:

- It does not violate the single responsibility principle of coding or break encapsulation.
- It allows the application's classes to act in a stand-alone manner and does not give too much reliance to a singular class.
- Calling the toString() methods from the other classes allows for less room for error, which could come from error in the wording of the individual print functions if the alternative design was chosen.
- This design further strengths the program's class hierarchy and reduces the number of errors that could come from outside classes.

A design considered for the SQL database included using different objects' UUIDs as primary keys and creating tables for Equipment, Material, Contract, Rental, and Lease. However, this design breaks normal form rules and overcomplicates the system. Instead, each object was given an internal Id number to use as the primary key and Equipment, Material, and Contract were put in the Item table with constraints. Rental and Lease were also put into the InvoiceItem table with constraints as well. This design was chosen because:

- It does not violate any normal form regulations.
- It does not use any external data (i.e., any data not generated by the database itself) as primary or foreign keys.
- It simplifies the Item table and makes it easier to delete and insert an Item.
- It also strengthens the program's hierarchy by using a single table inheritance design, which reduces the number of errors that may arise in the program.

3. Detailed Component Description

This section is a thorough description of the project. Each section will go in-depth and cover how the project works. Further subsections will cover the design of the Database and how it is tested; the class model and how it is tested; the database interface and how it is tested; and finally, the sorted list data structure and how it is tested.

3.1 Database Design

This database design closely follows the class structure (section 3.2) as it creates tables that represent the objects used in the invoice, while also creating a relationship between the tables so they can be used coordinated with each other. This creates a relationship between tables that also allows for compatibility between the database and the Java application through calling SQL queries.

Additionally, it also supports VGB's business model as it develops the database-backed invoice subsystem that keeps track of VGB's invoices and billing, which is what Ron Swanson, the company's CEO, requested. It does this while also keeping the same object-oriented approach that the java application has. It uses constraints in the Item table to determine if an Item is equipment, material, or contract. The InvoiceItem table also uses constraints to determine if equipment is purchased, leased, or rented, so that the invoicing and billing systems can determine how much to charge for certain items.

Furthermore, the database is designed this way to allow internal connections between tables and to prevent breaking normal form rules. Each table is given an Id as a primary key so that all foreign keys would use internal data instead of external data, allowing for no violation of normal form. A single table inheritance strategy is also taken on to make it easier to modify data in the tables. This allows for the database to model the Java application classes without breaking any normal form violations or over-complicating the database.

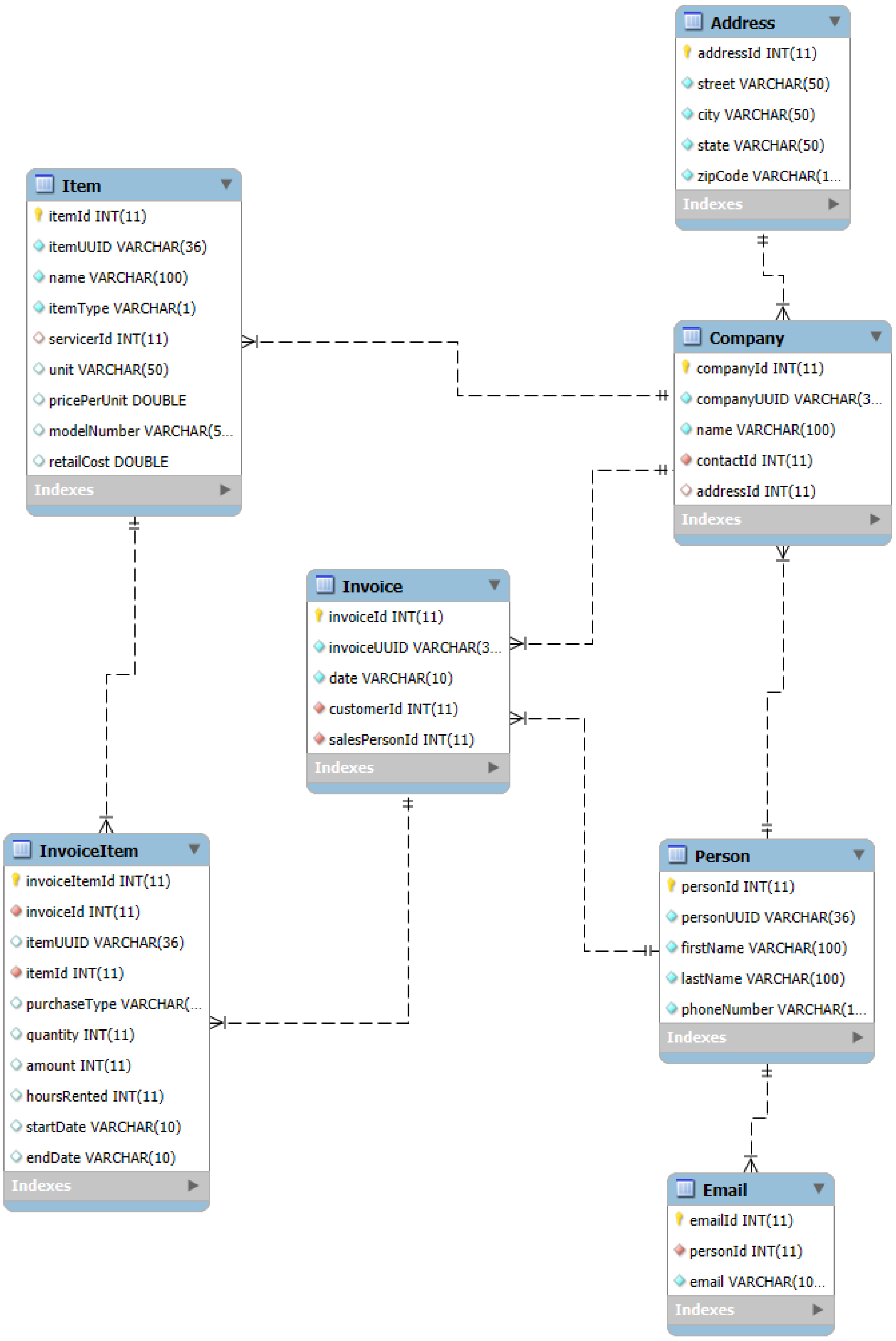


Figure 1: An ER diagram of the SQL database

3.1.1 Component Testing Strategy

The test cases for the SQL database have five primary goals. The first goal is to make sure the foreign keys work and that there is a relationship between tables. The next goals are to make sure that the database can correctly retrieve, insert, and delete data. Finally, its last goal is to make sure the data is generated properly through queries. Test data is generated by Mockaroo ("Mockaroo", n.d) for these test cases. These test cases take the form of several types of data being inserted into the database and twelve SQL queries. The SQL queries check to make sure the five primary goals can be completed and that there are no errors that can be created in the database (i.e., a sales person being able to make a sale to a company they own on an invoice).

There is a total of sixty-nine records in the database to test. Each table has a different number of records to test, with

- The Person, Company, and Address tables having ten records
- The Email table having eight records
- The Invoice table having five records
- The Item table having twelve records
    - Four records each for Material, Contract, and Equipment
- The InvoiceItem table having fourteen records
    - Four records each for Contract and Material
    - Two records each for Purchase, Lease, and Rental

This is a sufficient number of test cases as they make sure each function of the database is covered and repeated enough times to assume that their successes, in regard to test cases, are not anomalies. Additionally, when the program runs these tests, it correctly passes them all.

3.2 Class/Entity Model

Figures 2 and 3 show the different classes used in the program that do not leverage class hierarchy directly. Instead, these classes take on the roles of loading data, acting as test cases to make sure data can be inserted into objects, file writing, rounding, and generating the summary Reports. This design allows the primary classes to be independent, reusable from one another, and prevents them from breaking encapsulation. It also helps test the program to make sure there are no errors and to easily catch any that may arise.

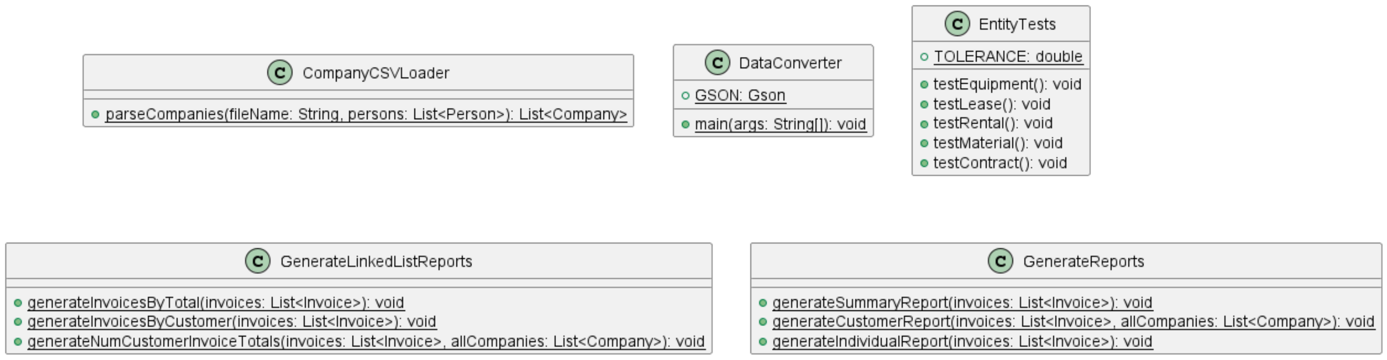

CompanyCSVLoader
parseCompanies(fileName: String, persons: List<Person>): List<Company>
DataConverter
GSON: Gson
main(args: String[]): void
EntityTests
TOLERANCE: double
testEquipment(): void
testLease(): void
testRental(): void
testMaterial(): void
testContract(): void
GenerateLinkedListReports
generateInvoicesByTotal(invoices: List<Invoice>): void
generateInvoicesByCustomer(invoices: List<Invoice>): void
generateNumCustomerInvoiceTotals(invoices: List<Invoice>, allCompanies: List<Company>): void
GenerateReports
generateSummaryReport(invoices: List<Invoice>): void
generateCustomerReport(invoices: List<Invoice>, allCompanies: List<Company>): void
generateIndividualReport(invoices: List<Invoice>): void

Figure 2: A Unified Modeling Language (UML) Diagram that represents classes that are not connected.

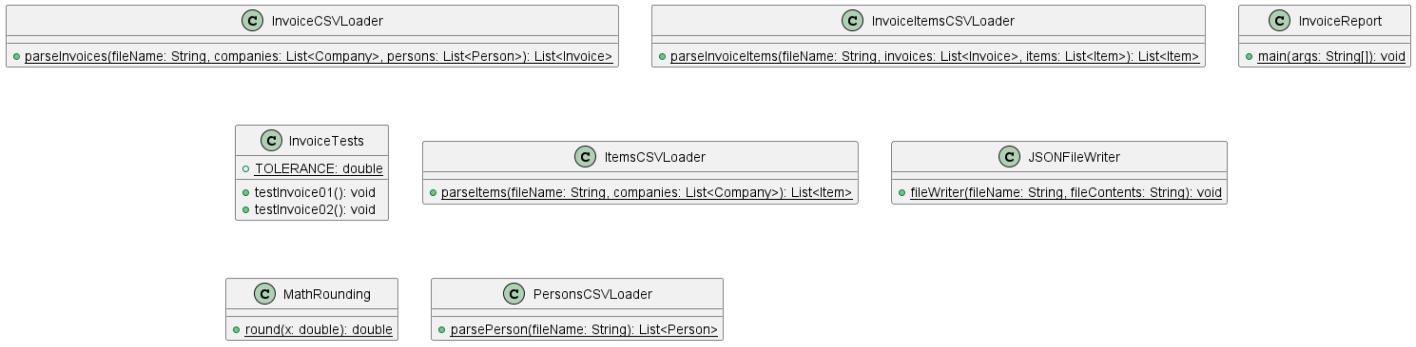

InvoiceCSVLoader
parseInvoices(fileName: String, companies: List<Company>, persons: List<Person>): List<Invoice>
InvoiceItemsCSVLoader
parseInvoiceItems(fileName: String, invoices: List<Invoice>, items: List<Item>): List<Item>
InvoiceReport
main(args: String[]): void
InvoiceTests
TOLERANCE: double
testInvoice01(): void
testInvoice02(): void
ItemsCSVLoader
parseItems(fileName: String, companies: List<Company>): List<Item>
JSONFileWriter
fileWriter(fileName: String, fileContents: String): void
MathRounding
round(x: double): double
PersonsCSVLoader
parsePerson(fileName: String): List<Person>

Figure 3: A UML Diagram that represents the rest of the classes that are not connected.

Figures 4 and 5 show the class hierarchy used to create an Item in the application and how it leverages inheritance. Item acts as an abstract class and a parent class to Contract, Equipment, and Material. Rental and Lease function as extensions of Equipment. This class hierarchy design has Item use abstract functions with the subclass deciding what is returned. This design allows for an easier creation or modification of an Item while also making it easier to apply the necessary taxes needed.

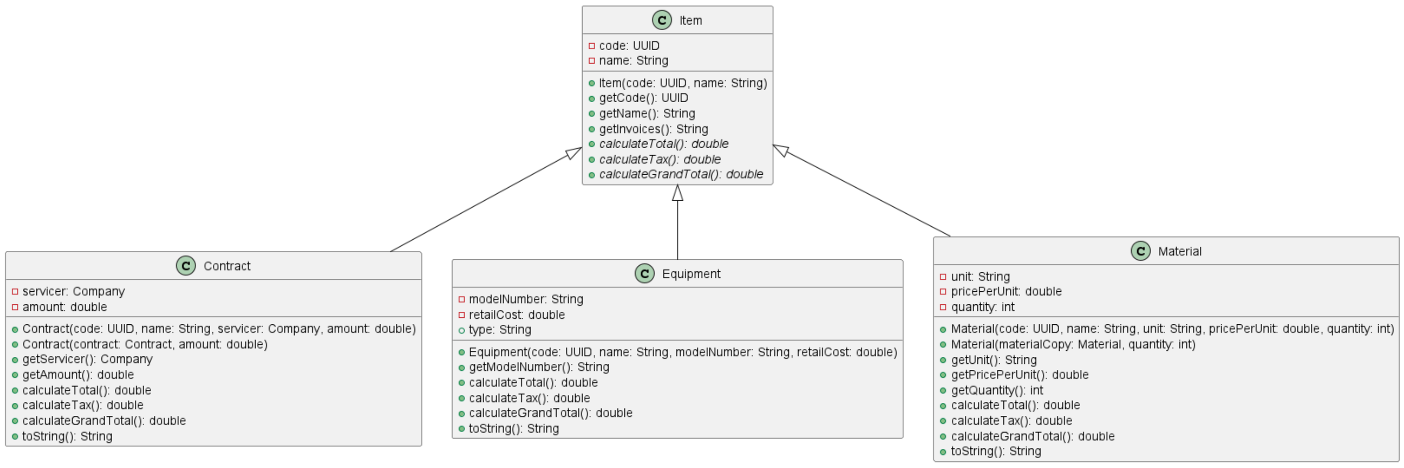


Figure 4: A UML Diagram that represents Item's class hierarchy

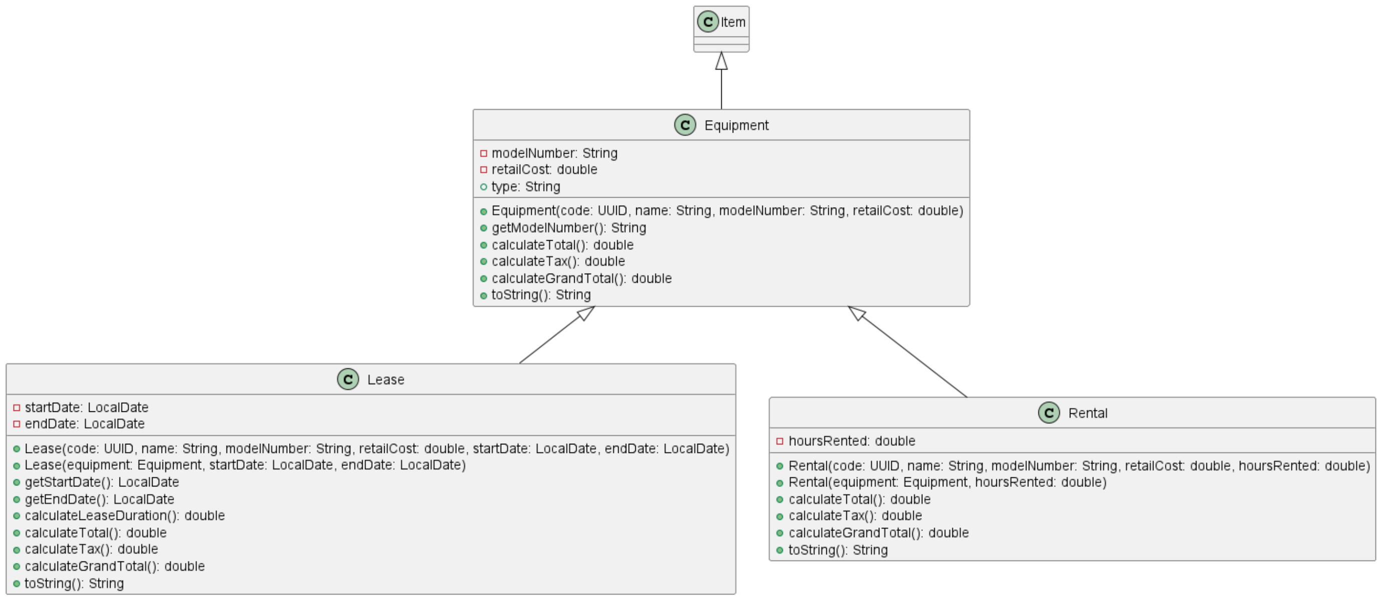

Item
Equipment
- modelNumber: String
- retailCost: double
~ type: String
+ Equipment(code: UUID, name: String, modelNumber: String, retailCost: double)
+ getModelNumber(): String
+ calculateTotal(): double
+ calculateTax(): double
+ calculateGrandTotal(): double
+ toString(): String
Lease
- startDate: LocalDate
- endDate: LocalDate
+ Lease(code: UUID, name: String, modelNumber: String, retailCost: double, startDate: LocalDate, endDate: LocalDate)
+ Lease(equipment: Equipment, startDate: LocalDate, endDate: LocalDate)
+ getStartDate(): LocalDate
+ getEndDate(): LocalDate
+ calculateLeaseDuration(): double
+ calculateTotal(): double
+ calculateTax(): double
+ calculateGrandTotal(): double
+ toString(): String
Rental
- hoursRented: double
+ Rental(code: UUID, name: String, modelNumber: String, retailCost: double, hoursRented: double)
+ Rental(equipment: Equipment, hoursRented: double)
+ calculateTotal(): double
+ calculateTax(): double
+ calculateGrandTotal(): double
+ toString(): String

Figure 5: A UML Diagram that represents Equipment's class hierarchy

Figure 6 shows the class hierarchy of the remaining classes and how they leverage inheritance. Unlike the previous figure which show Item being the parent class of a set of subclasses, these classes (Person, Company, Address, Invoice, Item) do not have a set parent class.

The Person class does not need data from any other classes to be created, but it does use its data in numerous other classes. Address acts in a similar manner, but its data is only used in Company. The Company class, on the other hand, relies directly on Person and Address to be created, as that is where it gets the Company's contact variable and its address variable. It also relies on Invoice to get a Company's invoices. Item acts in a similar manner to Person, where none of the other classes in the UML diagram are needed for it to be created. The Invoice class relies on Company, Person, and Item to be created. The Invoice class gets its customer variable from Company, its salesPerson variable from Person, and it gets the items on an invoice from Item.

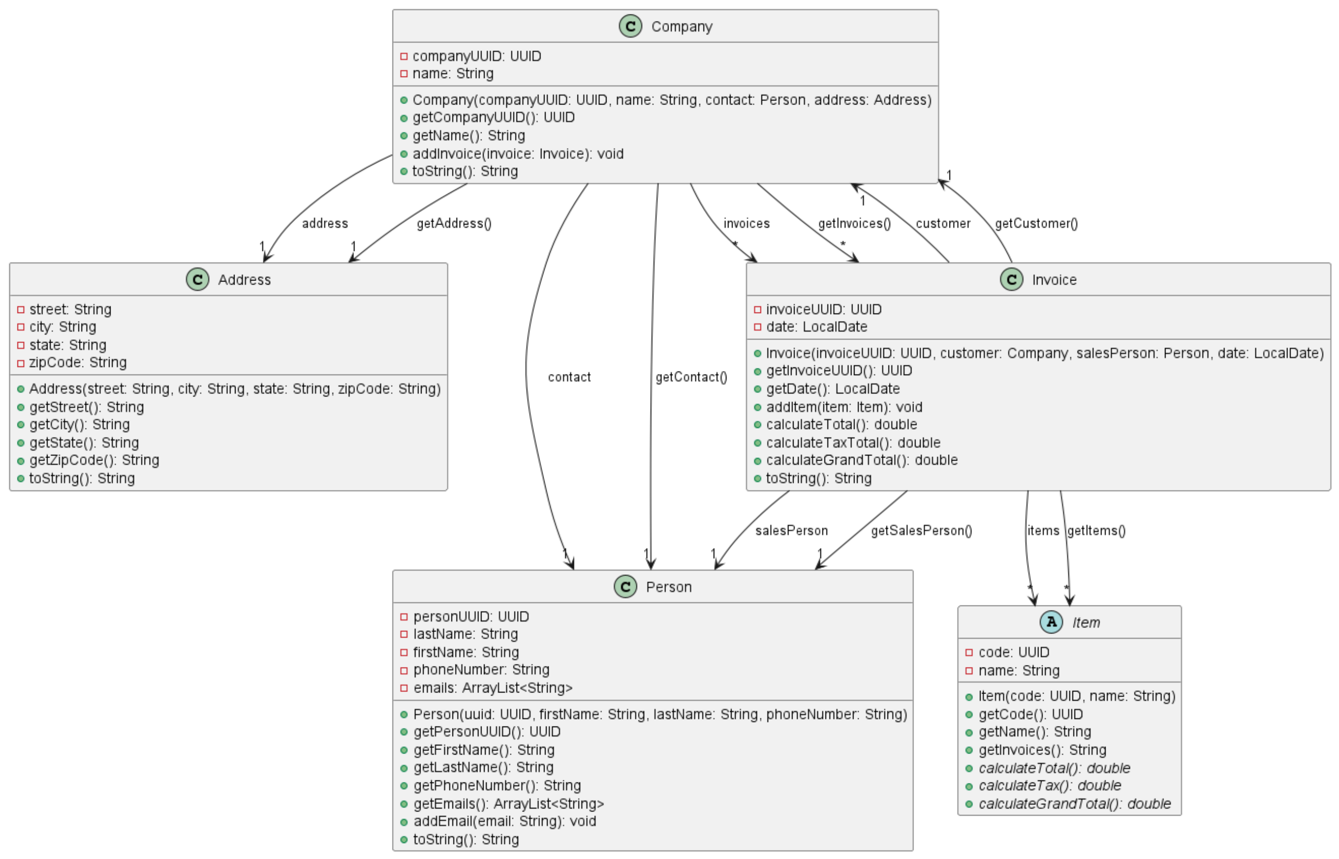


Figure 6: A UML Diagram that the other classes' hierarchy

Figures 7 and 8 show the use of Inheritance for the Contract class, which is a subclass of Item. The Contract class must have a servicer variable on it, which it inherits from the Company class. As previously stated, Company relies on inheritance from Address, Invoice, and Person to be created. Address creates the address variable, Person creates the contact variable, and Invoice retrieves all invoices created for a company.

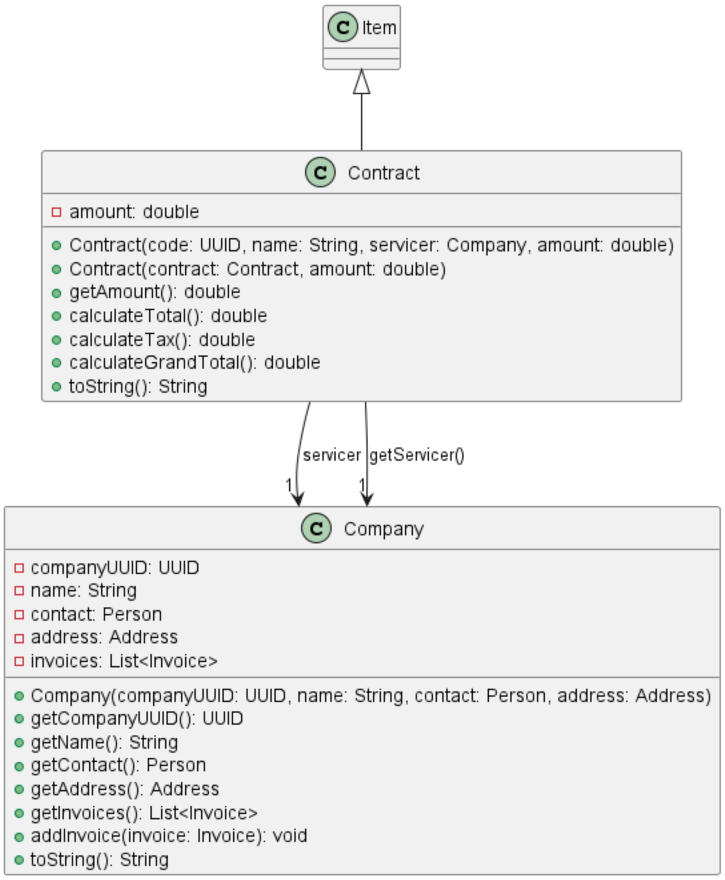


Figure 7: A UML Diagram that shows how Company and Contract connect

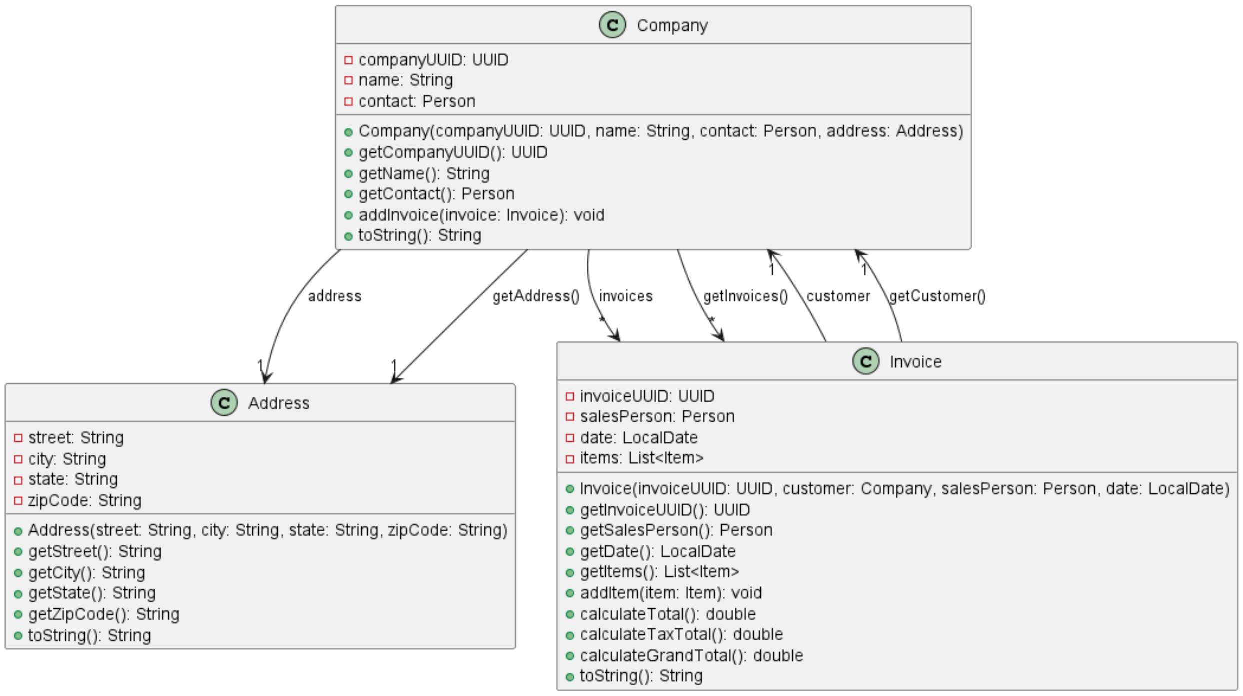


Figure 8: A UML Diagram that shows how Company connects to Invoice and Address

The figures above show the classes used in the Java application and break down their design. This design supports VGB's business model as it creates the application that works alongside the relational database to create the database-backed invoice system that keeps track of VGB's invoices. Classes are broken down into a hierarchy that splits the different types of Items and the different purchase types of Equipment, while also applying the necessary tax. This design also allows for an easier creation or modification for several types of invoice data.

This design's approach also leverages inheritance to create an Object-Oriented approach that can also be used interchangeably alongside a relational database. Figures 3 through 8 split up the diagram to show the inheritance and class hierarchy, but Figure 9 shows how all the classes are connected via inheritance. Despite these classes all being connected via a hierarchy, they are still independent of each other and are not overly reliant on one another.

Additionally, this Java application is designed to work alongside an SQL database. The application uses its relationship between classes to allow compatibility to read in from Comma Separated Values (CSV) files or the SQL database.

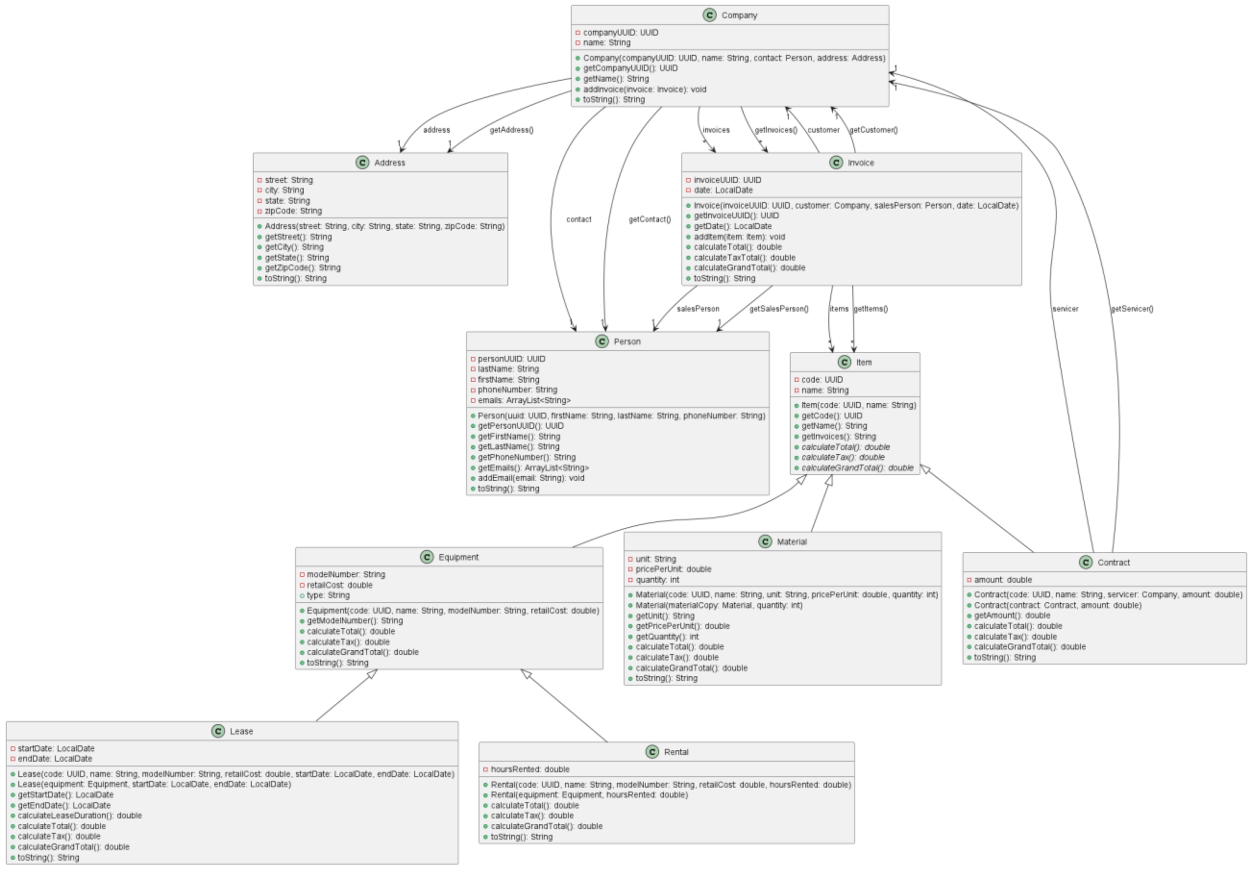


Figure 9: A UML Diagram that shows all the classes put together in a hierarchy

3.2.1 Component Testing Strategy

An external testing environment provided by the client generates a vast majority of the test cases for this program. However, to ensure minimal failure, test cases were made alongside the program to test it. There are several of these test cases, each with a different intended purpose and they are tested through a JUnit testing suite. When the application runs all these test cases, it passes them all. These test cases include:

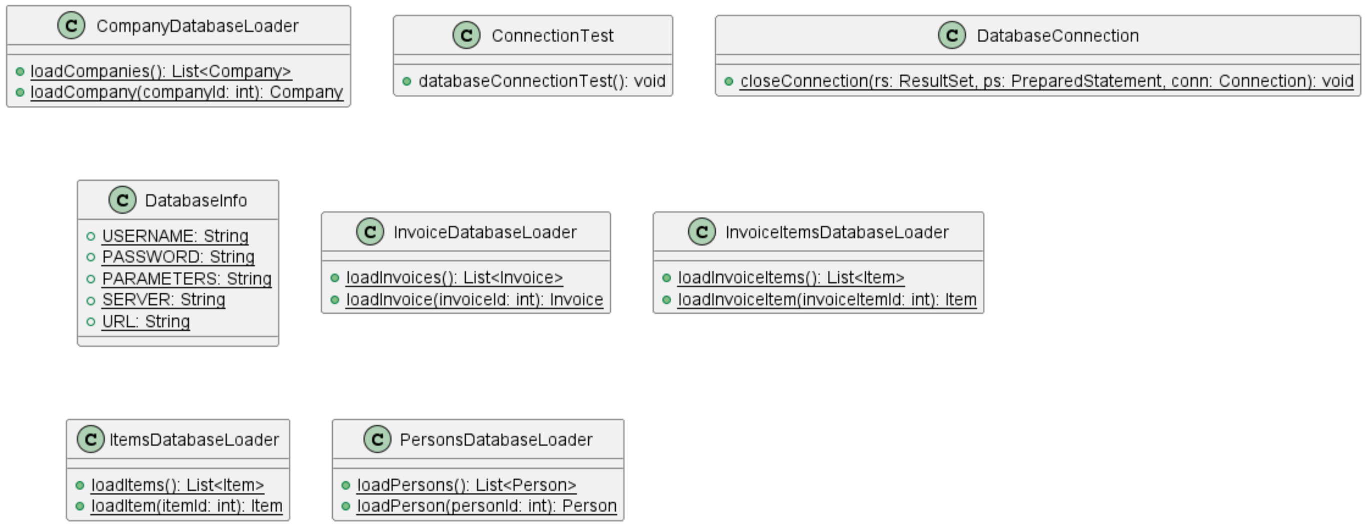
- Test cases that make sure that data for the Items, Company, and Persons classes are properly read in and output.
- Several test cases for the Invoice, Items, Equipment, Material, Contract classes that read in several different files for invoices. One of these test cases contains two data types that represent items and another contains three. These test cases represent all five types of items and their purchase types that can be used.
- Test cases were also designed for generating summary reports of the Invoice class. They make sure that the data files are loaded in from InvoiceCSVLoader.java and InvoiceItems.CSVLoader and correctly output the three reports in GenerateReports.java. These test cases focus on making sure that the three reports that are output are factually correct and read in the data correctly from the invoices.

3.3 Database Interface

Data is loaded in from the SQL database into Java objects via an API. It creates and uses a JDBC connection to the SQL database. Additionally, each Object fetches the data via an SQL select query and uses Prepared Statement and Result Set to execute queries and store it properly. The foreign key Ids used in the SQL database, similar to the UUID used in the csv files, allow it to follow the class hierarchy used and create relationships between Objects. Figure 10 shows the classes used to accomplish this design:

- DatabaseConnection prevents repetitive code and closes the database connection in each of the loader classes.
- DatabaseInfo stores the information to connect to the SQL database
- ConnectionTest test to see if a connection is made to the SQL database.
- CompanyDatabaseLoader connects to the database, gets the variables used for Company and Address in the SQL database and stores them into object variables for the Company class.
- PersonsDatabaseLoader also connects to the database and inserts data into variables, except it does it for the Person class
- InvoiceDatabaseLoader retrieves data from the Invoice table and inserts it into variables for Invoice.

- ItemsDatabaseLoader also connects to the database and inserts data into variables, except it leaves some values at 0 (i.e., hoursRented) as those variables are out of its scope.
- These values are then replaced with real values by InvoiceItemsDatabaseLoader, which gets its data from the InvoiceItem table and fills in the remaining variables.

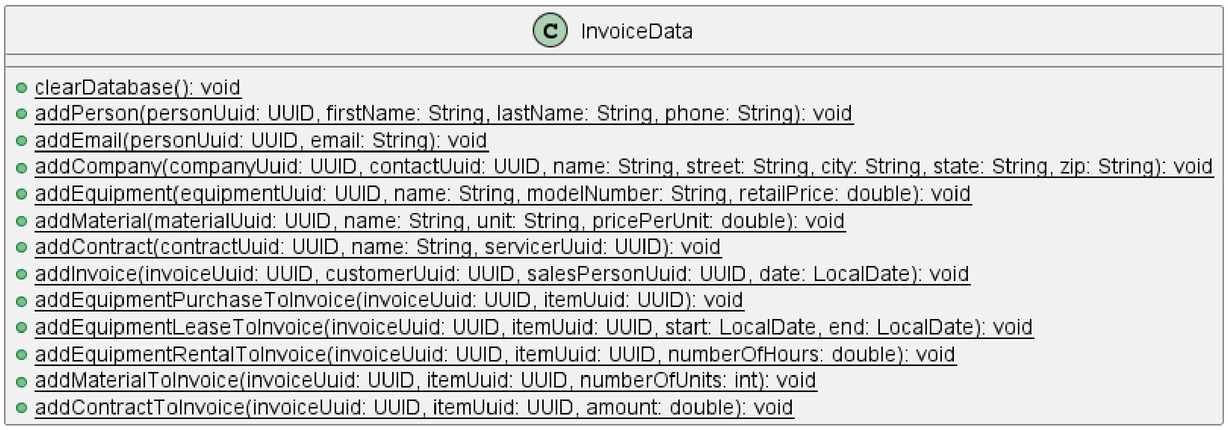


CompanyDatabaseLoader
loadCompanies(): List<Company>
loadCompany(companyId: int): Company
ConnectionTest
databaseConnectionTest(): void
DatabaseConnection
closeConnection(rs: ResultSet, ps: PreparedStatement, conn: Connection): void
DatabaseInfo
USERNAME: String
PASSWORD: String
PARAMETERS: String
SERVER: String
URL: String
InvoiceDatabaseLoader
loadInvoices(): List<Invoice>
loadInvoice(invoiceId: int): Invoice
InvoiceItemsDatabaseLoader
loadInvoiceItems(): List<Item>
loadInvoiceItem(invoiceItemId: int): Item
ItemsDatabaseLoader
loadItems(): List<Item>
loadItem(itemId: int): Item
PersonsDatabaseLoader
loadPersons(): List<Person>
loadPerson(personId: int): Person

Figure 10: A UML Diagram that shows the classes used to insert the SQL data into Java Objects

Bad data, which is data that does not work properly with the application, is removed to keep data integrity. Duplicate and null values for an item are considered bad data in this scenario. Examples of bad data are UUIDs and Ids that do not match to anything, types of Items with data inserted into the wrong fields (Equipment with fields for Material), data with null values where they should not be null, etc. This data is caught via try-catch statements and not ported over to the application. Good data, on the other hand, is data that works properly with the application and is not removed to keep data integrity (i.e., UUIDs and IDs that connect to other objects to create relationships, Items not having null data for necessary variables, etc.).

Data modification is set up similar to DatabaseLoader but it uses SQL insert statements instead of select statements. If statements are used prior to modifying the data to make sure the correct fields are not null. If they pass the if statements, an insert SQL query is called to insert the data, similar to the select statements to pull in data. If they fail, an SQL exception is given. Furthermore, try-catch statements surround these functions as well, which are used to execute the queries using PreparedStatement, setters, and keys. This process is visualized by Figure 11.



```
InvoiceData
clearDatabase(): void
addPerson(personUuid: UUID, firstName: String, lastName: String, phone: String): void
addEmail(personUuid: UUID, email: String): void
addCompany(companyUuid: UUID, contactUuid: UUID, name: String, street: String, city: String, state: String, zip: String): void
addEquipment(equipmentUuid: UUID, name: String, modelNumber: String, retailPrice: double): void
addMaterial(materialUuid: UUID, name: String, unit: String, pricePerUnit: double): void
addContract(contractUuid: UUID, name: String, servicerUuid: UUID): void
addInvoice(invoiceUuid: UUID, customerUuid: UUID, salesPersonUuid: UUID, date: LocalDate): void
addEquipmentPurchaseToInvoice(invoiceUuid: UUID, itemUuid: UUID): void
addEquipmentLeaseToInvoice(invoiceUuid: UUID, itemUuid: UUID, start: LocalDate, end: LocalDate): void
addEquipmentRentalToInvoice(invoiceUuid: UUID, itemUuid: UUID, numberOfHours: double): void
addMaterialToInvoice(invoiceUuid: UUID, itemUuid: UUID, numberOfUnits: int): void
addContractToInvoice(invoiceUuid: UUID, itemUuid: UUID, amount: double): void
```

Figure 11: A UML Diagram of the InvoiceData class to modify data

3.3.1 Component Testing Strategy

Test data is designed to test the ability to load the data from the SQL database and produce reports. This test data is inserted into the SQL database and through the JDBC API, it is pulled into the Java application to use it as the data for the three different invoice reports. Since the data in the SQL database is the same as the data in the CSV files, the test cases include comparing the outputs from both to make sure they match. Additionally, a test case to make sure the program is connecting to the SQL database properly is implemented. Each object (i.e., Person, Company, Item, InvoiceItem, Invoice) has five of its own test cases. Other parts of the application (i.e., Rental, Lease, Equipment, Material, Contract, and Purchased) have four test cases each. This number of test cases gives each function enough tests to know that their success is not an anomaly.

The modification of data is tested by an external testing environment provided by the client, which generates its own data. The external testing environment displays the program's outputs with the expected output to see if they match and pass the tests. Comparing both outputs results in them being identical, which means the application passes the tests provided by the external testing environment.

3.4 Design & Integration of a Sorted List Data Structure

Figure 12 shows a visualization of the Sorted List Data Structure's design. The SortedList data structure is a node-based linked list. The data structure's interface consists of functions for inserting, getting, deleting, and finding the size. Using these functions, the data structure stores elements and inserts new elements based on the result of a set of Comparator<> functions, which are shown on the bottom row of Figure 13.

The Node class allows it the ability to easily insert and delete new nodes/indexes into the list. This allows the list to only add new items in the indexes where they belong, which means the list will never be out of order. Since there are three different summary reports, three different Comparators are introduced. The class to generate reports is shown in Figure 14. The first function compares the total of an invoice (and uses the invoice UUID to break any ties), the second compares the customer names (and uses the UUID of the invoice to break any tie breakers), and the third compares the total of all of a company's invoices (using the company UUID as a tie breaker). Due to this, the data structure is able to maintain order. The ability to add new elements to the list is done by traversing the list and using the comparator to make sure that the correct position is found in the sorted list and then inserting the element. Deleting elements is done by traversing the linked list until it finds the matching element and deleting its node from the list.

This design is generic to allow the data structure to store any type of object that uses the three Comparators. This allows the LinkedList class to be reused for the invoice total, invoice customer name, and number of invoices for a company, so that it only needs to be implemented once.

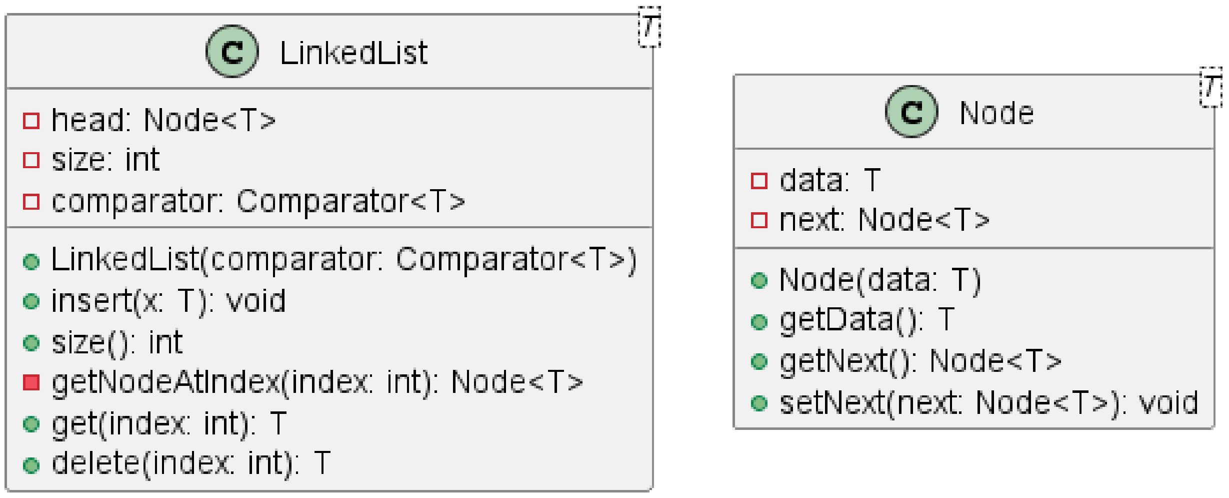
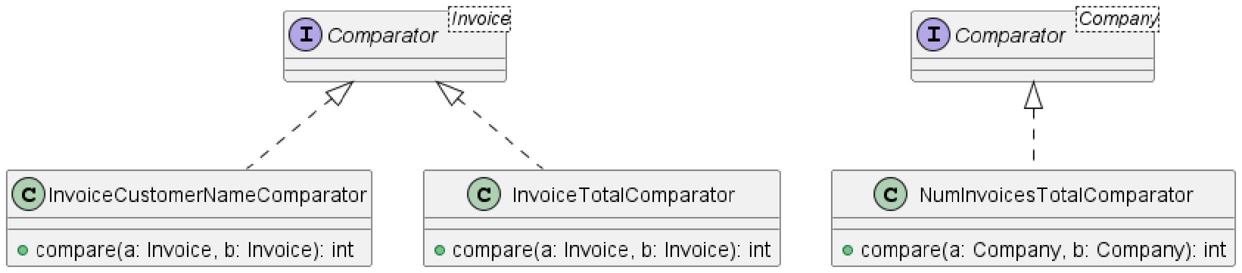
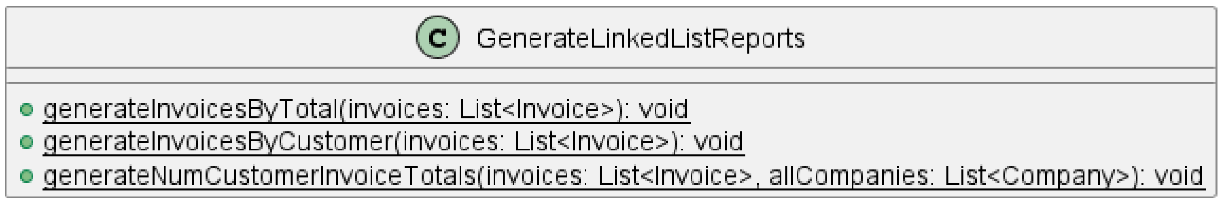


Figure 12: A UML Diagram of the Sorted List Data Structure



I Comparator
Invoice
I Comparator
Company
C InvoiceCustomerNameComparator
compare(a: Invoice, b: Invoice): int
C InvoiceTotalComparator
compare(a: Invoice, b: Invoice): int
C NumInvoicesTotalComparator
compare(a: Company, b: Company): int

Figure 13: A UML Diagram of the Comparators Used



C GenerateLinkedListReports
generateInvoicesByTotal(invoices: List<Invoice>): void
generateInvoicesByCustomer(invoices: List<Invoice>): void
generateNumCustomerInvoiceTotals(invoices: List<Invoice>, allCompanies: List<Company>): void

Figure 14: A UML Diagram of the class for generating the reports using a Linked List

3.4.1 Component Testing Strategy

An external testing environment provided by the client tests this design. This testing environment loads in several different sets of SQL data and produces an expected and actual output. The primary form of testing is through comparing both outputs and checking for any inconsistences or errors. The testing environment has four different SQL databases that it loads in and tests. Each one contains several different invoices with different companies/customers, which makes sure that the application can properly support the comparator. It also contains multiple invoices for a singular company/customer. This tests to make sure that the application can properly break comparator ties with the UUID of the object being sorted.

The testing environment accomplishes this by clearing the previous SQL database and loading in a new database. It then produces three summary reports that sort by invoice total, customer name, or the total of all of a customer's invoices. These four different test cases the external environment provides are a sufficient number of test cases as each test cases provides numerous amounts of unique data for the Linked List to sort. Having four different test cases also makes sure that there are no anomalies when it comes to the expected and actual outputs matching. When the application is run through the external testing environment, it passes all four tests with the outputs matching.

4. Changes & Refactoring

During the development of JDBC Database connection, a LocalDate-JSON adapter was implemented into DataConverter to prevent an issue that came about due to LocalDate variables being used. This issue, before it was fixed, prevented the application from writing to JSON files.

Bibliography

[1] *Mockaroo.com* (n.d.). Retrieved March 26, 2025, from https://www.mockaroo.com/

[2] Gupta, L. (2024, October 15). Number of Days between Two Dates in Java.

Retrieved from https://howtodoinjava.com/java/date-time/calculate-days-between-dates/

[3] Bourke, C. (2021). *Computer Science II* (3rd ed). University of Nebraska-Lincoln