Very Good Building & Development Company

Invoice Subsystem

Yin Po Po Aung

yaung3@huskers.unl.edu

Rometh Samarasinghe
rsamarasinghe2@huskers.unl.edu

University of Nebraska—Lincoln

Spring 2025
Version 6.0

This document provides the technical design for the Invoice Subsystem of Very Good Building & Development Company (VGB), Ron Swanson's company.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Yin Po Po Aung Rometh Samarasinghe	2025/02/14
2.0	Second draft of this design document: modify the introduction that includes scope of the document and add some information to overall design description, alternative design options and database design	Yin Po Po Aung Rometh Samarasinghe	2025/02/28
3.0	Third draft of the design document: Add the table for calculating the price based on the transaction type to introduction part, add MYSQL database structure information and ER diagram and finally add UML diagrams for the class model separately to be clear	Yin Po Po Aung Rometh Samarasinghe	2025/03/28
4.0	Fourth draft of design document: Modify the overall design description, alternative design description and component testing strategy. Add the information for database interface and updated the UML diagram 2.6	Yin Po Po Aung Rometh Samarasinghe	2025/04/11
5.0	Fifth draft of design document: Modify overall design description, class model, uml diagrams and component testing strategy. Add the information for design & integration of a Sorted List Data Structure and additional material	Yin Po Po Aung Rometh Samarasinghe	2025/04/25
6.0	Final draft of design document: Review all the parts and write database design and class entity/model with updated ER diagrams. Add more information about Design & Integration of a Sorted List Data Structure and additional material	Yin Po Po Aung Rometh Samarasinghe	2025/05/09

Table of Contents

1. Introduction

This document provides the technical design for the Invoice Subsystem of Very Good Building & Development Company (VGB), led by Ron Swanson. VGB is a construction company involved in various facets of the industry, including general contracting, subcontracting, and the sales, leasing, and rental of construction equipment and materials.

Currently, the company relies on spreadsheets and physical records, which necessitates the modernization of its processes through a database-backed system. To address this need, the system is designed as an object-oriented application, written in Java, which integrates data representation and electronic data interchange (EDI) using JDBC.

The report system is designed to generate three types of reports which are a summary of all invoices with totals, a summary of invoices grouped by customers, and a detailed report for each individual invoice. Furthermore, the system handles different types of transactions, including purchases, rentals, and leases, each with its own tax calculation model.

Below, the tax rules and final pricing models for each transaction type are outlined to ensure consistency and accuracy in invoicing.

Equipment Type	Calculation	Tax Calculations	Total Price Calculations
Equipment Purchase	Price of item	Tax at 5.25%	Total = Price + Tax
Equipment Lease	Amortized price over lease period	Tax at flat rate of $1,500 (if total > $12,500)	Total = Amortized Price * 1.5 + Flat Tax
Equipment Rental	Per-hour charge, (0.1% of item price per hour)	Tax at 4.38%	Total = Rental Charge * Hours + Tax
Material	Price per unit * Quantity	Tax at 7.15%	Total = Price * Quantity + Tax
Contract	Flat price (no tax on contract)	No tax	Total = Contract price

1.1 Purpose of this Document

This document describes the design and implementation of the Invoice Subsystem for Very Good Building & Development Company (VGB). The purpose of this document is to outline the initial data representation and electronic data interchange (EDI) approach. This

technical design document specifies the architecture and implementation details of VGB's data representation and electronic data interchange systems. It serves as the authoritative reference for the system's data models and file processing capabilities.

1.2 Scope of the Project

This section covers the features and functionalities of the invoicing subsystem under the inventory management supersystem of VGB. The **Invoice subsystem** is responsible for keeping track of all invoices, billing and producing detailed reports. The core functionalities of the subsystem include invoice generation, data storage, and structured reporting.

This Subsystem is focused on designing and implementing a Java-based invoice subsystem that can execute CRUD queries, create object representations of persons, companies, and items and serialize these objects into text reports for interoperability. Furthermore, it involves a proper class hierarchy with appropriate inheritance relationships, implementing core functionality for invoice calculations, and developing comprehensive testing suites.

- Implementing calculation methods for Equipment purchases, leases, and rentals
- Implementing calculation methods for Material purchases
- Implementing calculation methods for Contract services
- Creating an Invoice class to aggregate items and calculate totals
- Developing JUnit test suites
- Testing queries with edge cases (fraud, duplicates, etc.)

Extending functionality and producing reports, this outputs three well-formatted and detailed text files:

- Summary report: Summary of all invoices along with a few totals
- Invoice Report: Details for each individual invoice
- Customer Summary: Summary for each customer

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Entity: A business object representing a real-world concept such as Equipment, Material, or Contract

Optimized CRUD Operations: More efficient relationships for seamless Create, Read, Update, and Delete functions

Data Integrity: Constraints and validation mechanisms enforce high-quality data

1.3.2 Abbreviations & Acronyms

VGB: Very Good Building and Development Company

UUID: Universally Unique Identifier which is a 128-bit identifier used to uniquely distinguish objects within the system.

XML: Extensible Markup Language, which is a structured format, used for encoding documents in a machine-readable manner.

JSON: JavaScript Object Notation which is a lightweight data interchange format which is easy for humans to read and write and easy for machines to parse and generate.

JDBC: Java Database Connectivity

2. Overall Design Description

The Invoice Subsystem is designed to handle data processing, execute SQL CRUD queries and produce three reports. The system is built around several key components which include data models, file processing, serialization services, and transaction functionality. At its core, the system utilizes a foundational layer for data management, with classes such as Person for contact information, Company for business entities, and an Item hierarchy that supports equipment, materials, and contracts.

Additionally, the Invoice class models the core structure of a sales transaction. It includes a UUID for tracking, a Company object to represent the customer, a Person object for the salesperson, and a list of InvoiceItem objects that detail the items involved in the transaction. It also provides methods to calculate the subtotal, tax, and grand total for the invoice. The design cleanly separates responsibilities by referencing external data models like Company, Person, and Item. The InvoiceItem class links an individual Item to its corresponding Invoice. It acts as a bridge between the invoice and its contents, allowing the system to associate multiple items with a single transaction. The Expenses interface involves financial computations, including methods for calculating subtotals, taxes, and total amounts. It extends its functionality to Item and its subclasses.

The subsystem executes queries using JDBC, while also incorporating data validation and error handling to ensure data integrity. Serialization services enable the system to output JSON and XML data objects with consistent formatting if necessary. These subclasses handle specialized calculation methods for different transaction types and tax rules. The system also includes testing frameworks to validate both individual entity calculations, multi-item invoice totals and has been edge tested with empty invoices and handle duplicate transactions under the same invoice.

Following that, the system is extended to retrieve and manipulate data using a MySQL database. The core business logic, including the driver class, remains largely unchanged. A new database interface layer has been introduced, leveraging JDBC to handle all database interactions.

The DataLoader class takes on the responsibility of reading data from the database and populating Java objects, while the DataFactory class is tasked with constructing specific Java objects through SQL queries. The InvoiceData class which provides a set of utility

methods to add, update, delete, and retrieve data serves as the primary API for CRUD operations. Additionally, the ConnectionFactory class manages secure creation and closure of database connections.

Besides, a Binary Search tree is implemented to search and sort the report values. The elements in each SortedListBST are sorted according to the provided comparator. Specifically, invoices can be sorted in descending order of total amount or ascending order of customer name. Companies are sorted in ascending order of total invoice amounts. The ordering is fully determined by the comparator passed when constructing the list.

2.1 Alternative Design Options

A hybrid approach, combining an abstract base class for shared functionality and interfaces for specialized behaviors, is chosen over a simple inheritance model. This approach improves code reusability allowing different entity types to implement unique behaviors without duplicating code.

A subclass-based system is chosen instead of an enum-based type system with switch statements. This design ensures better encapsulation and extensibility, making it easier to introduce new item types without modifying existing logic.

Method overloading is chosen over the strategy pattern for cost calculation. While the strategy pattern provides separation of concerns, method overloading offers a simpler yet extensible solution, making transaction processing more efficient without unnecessary complexity.

The creation of a Lease and a Rental abstract class is considered but due to added complications to inheritance with Equipment, and the excess implementation it would be facing if Lease and Rental were interfaces, it is decided to have them as Objects and then used by Equipment class.

The DataFactory class is designed to ensure clean query execution and resource management. Of course, it is possible to have Connections opened and closed in each method but considering scalability it is decided to have a separate class to abstract away SQL query execution logic and using the result set in the DataLoader methods.

3. Detailed Component Description

This section provides an overview of the database design, class structures and their role in the system.

3.1 Database Design

The database is structured to manage invoices, items, companies, and their relationships. It consists of the following main tables:

Database Structure

The database consists of multiple tables to represent entities and their relationships. The primary tables include Company, Person, Email, Address, State, ZipCode, Invoice, InvoiceItem, and Item. Each table is structured with appropriate data types, primary keys, and foreign key constraints to maintain referential integrity.

- Company Table: Contains general information about companies, including a reference to a primary contact person and address details.
- Person Table: Stores details of individuals associated with the system, including first name, last name, and phone number.
- Email Table: Maintains email addresses linked to individuals in the Person table.
- Address Table: Contains detailed location information, including street, city, state, and zip code.
- State Table: Stores state information to ensure geographic data normalization.
- ZipCode Table: Holds zip codes associated with states for efficient address management.
- Invoice Table: Represents sales transactions, linking customers (companies) and salespersons (employees) with invoice records.
- InvoiceItem Table: Maintains details of items involved in each invoice transaction, including quantity, price, and rental periods.
- Item Table: Stores information on various products and services available for purchase or rental.

Database Key Relationships

1. Person Relationships

- Multiple Emails: One person can have multiple email addresses.
- Company Contacts: A single person can be a contact for multiple companies.
- Sales Tracking: Same person can be a salesperson across different invoices.

2. Company Relationships

- Single Primary Contact: Each company has one primary contact person.
- Shared Address: Multiple companies can share the same physical address.
- Invoice Tracking: Multiple invoices can be associated with a single company.

3. Invoice Relationships

- Multiple Items: One invoice can contain multiple invoice items.
- Customer Linkage: Each invoice links to a specific customer company.
- Salesperson Tracking: Each invoice is associated with a specific salesperson.

4. InvoiceItem Relationships

- An intermediate class for joining Items and Invoices
- Contains an Invoice that maps to a list of items

5. Address Relationships

Geographical Normalization:

The design follows the third normal form (3NF) principles to reduce redundancy.

- Each address is linked to a specific state
- Each address is linked to a specific zip code
- Company Connection: Multiple companies can use the same address

6. Item Relationships

- Invoice Items: The InvoiceItem table establishes this many-to-many relationship between the Invoice and Item tables. This enables the system to track which items are part of which invoices, allowing for detailed reporting and financial tracking
- Detailed Tracking: Each item can have multiple attributes (type, model, pricing)

7. Duplicate entry handling and Nullable fields

- All primary entity tables incorporate a UUID field designated as the unique key to prevent identical records with the same identifier. However, this implementation does **not** prevent semantically duplicate entries (i.e., records with identical values but different UUIDs). For example, a user could insert multiple rows with the same Email, ZipCode, Address, or State, as no additional uniqueness constraints are applied to these attributes.

- For normalization and operational flexibility, the following tables permit duplicate entries:

    - Address

    - ZipCode

- Email
- State

- Nullable fields in Invoice and InvoiceItem tables serve a functional purpose:
  - These fields allow differentiation between item types when inserting via JDBC.
  - No additional constraints exist beyond logical binding between Item.itemType and InvoiceItem.typeEquipment.

  This design grants flexibility in handling multiple item categories.

- The UUID column in InvoiceItem is nullable by design. It serves as an **optional lookup** key to associate an InvoiceItem with an Invoice. If the UUID is absent, the system defaults to using the InvoiceItem.id. This ensures:
  - Compatibility with factory-generated models
  - Consistency in handling entities whether a UUID is present or not

  The ER diagram below visually represents the database structure, showing the relationships between entities.

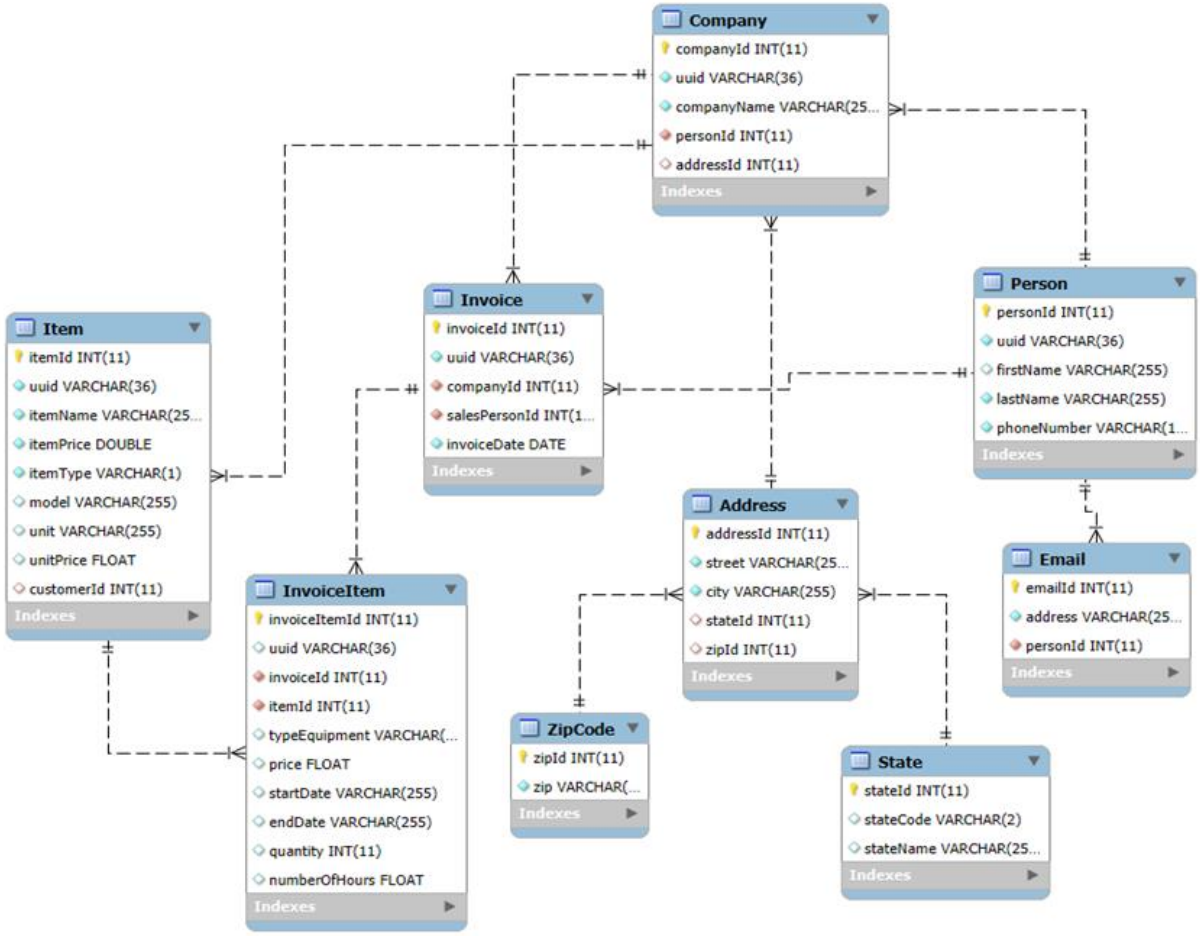

Figure 1: Database Design

3.1.1 Component Testing Strategy

SQL queries are used for direct database testing, including 12 core test queries that verify person attributes, email relationships, invoice details, customer purchasing patterns, and data integrity validations.

3.2 Class/Entity Model

The subsystem is based on implementing a foundational layer for data handling with these key components:

Data Models

- Person class for contact information
- Company class for business entities
- Item hierarchy for equipment, materials, and contracts

Database management

- Connection factory to instantiate connections to database
- DataLoader class to load tables and records using Factory classes
- DataMapper Interface implemented by Factory classes to be utilized when dynamically loading data with SQL queries

Class Hierarchy

- Interface base classes to define common behavior
- Abstract subclasses for specific entity types (Equipment, Material, Contract)
- Invoice-related classes to manage collections of items and calculate totals

Testing Framework

- EntityTests for validating individual entity calculations
- InvoiceTests for validating multi-item invoice calculations
- Test cases covering all five item types and various transaction scenarios

The project branches out to report printing functionalities utilizing the polymorphic behaviors implemented, this functionality is encapsulated into InvoiceReport class. It uses 3 methods to format 3 different reports and writes the report to a file, and one method to output to a text file.

Below is a breakdown of the key components in the diagram:

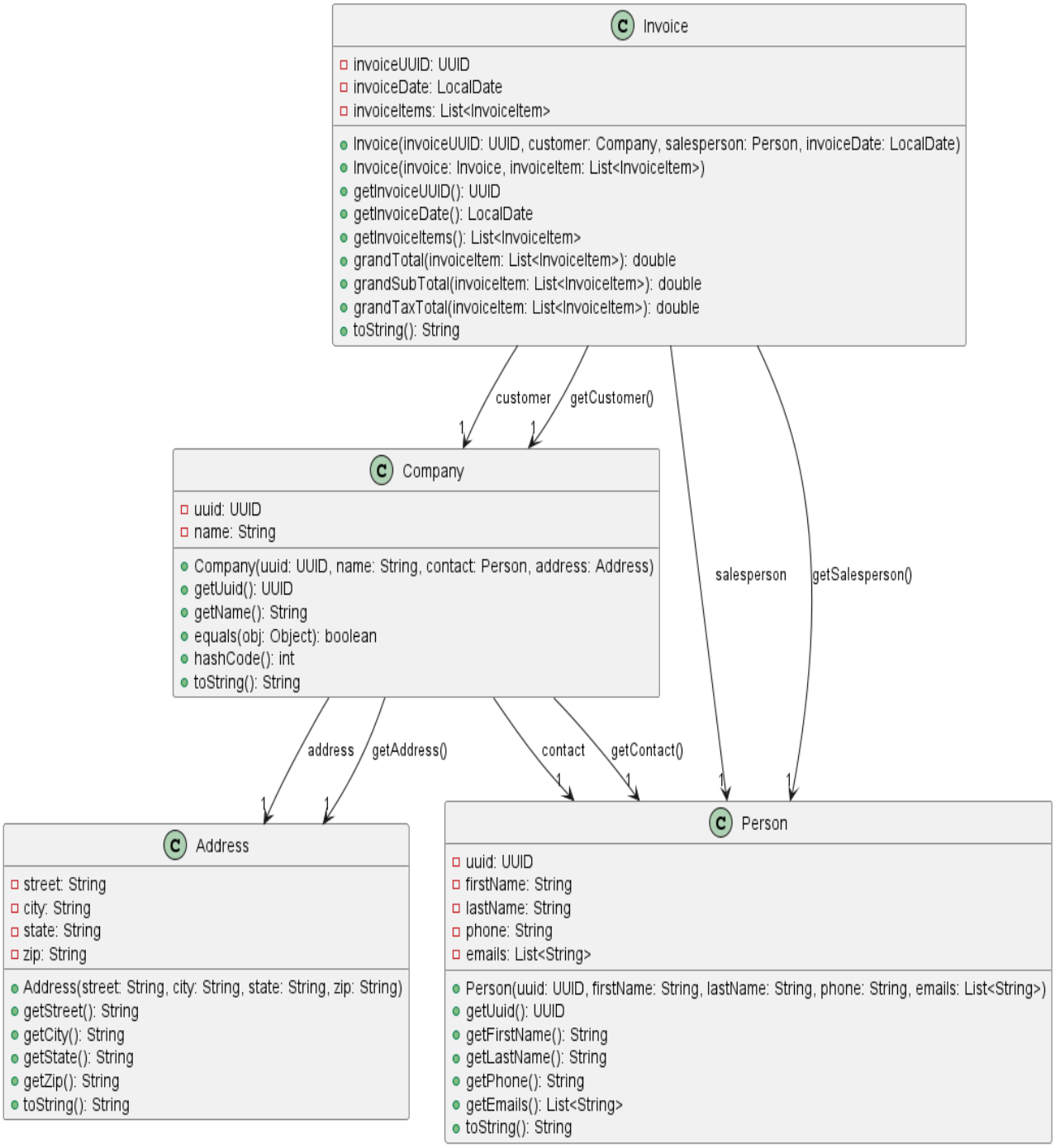

Figure 2.1 : Class Structure

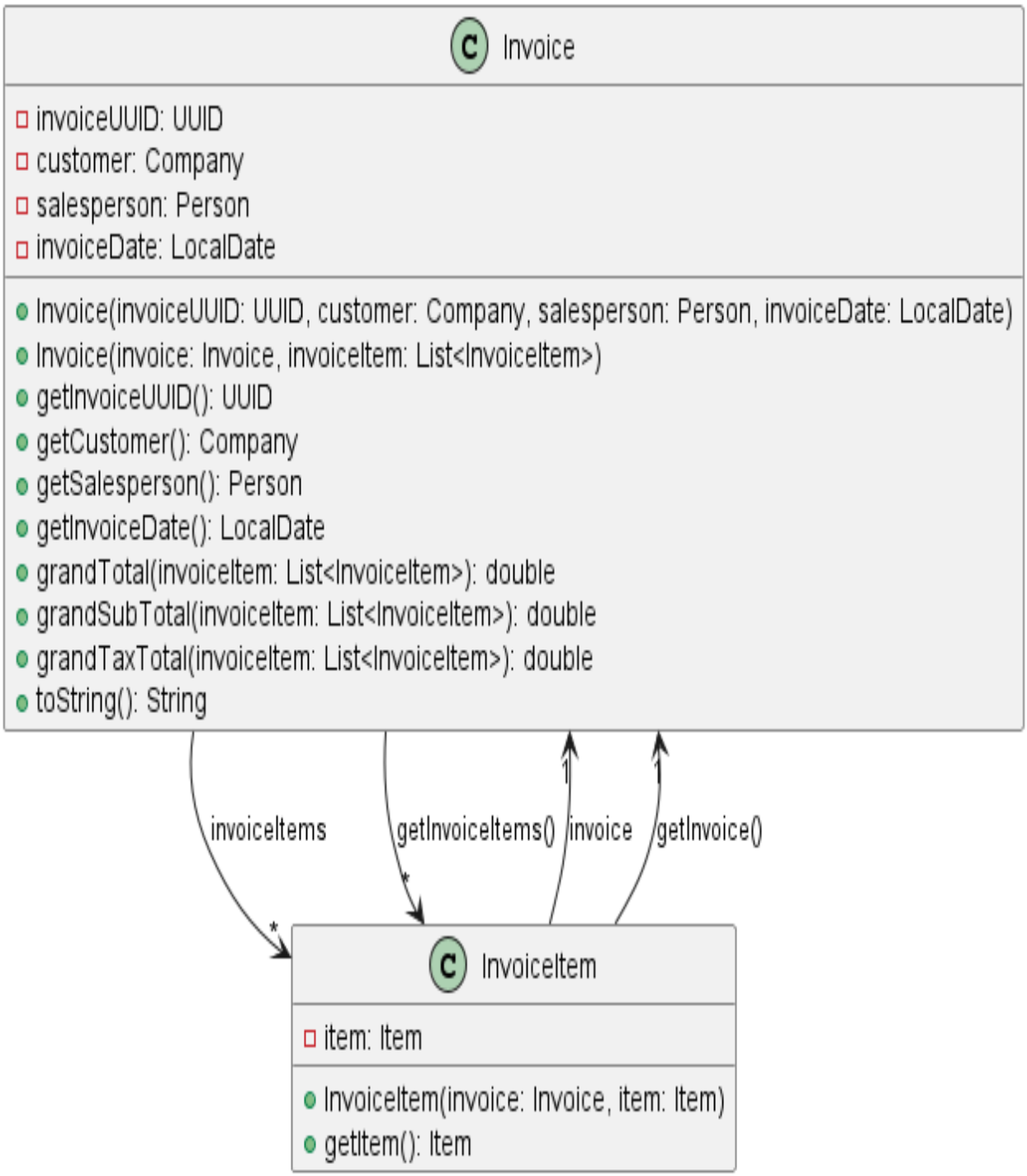

Figure 2.2 : Class Structure

Figure 2.3: Class Structure

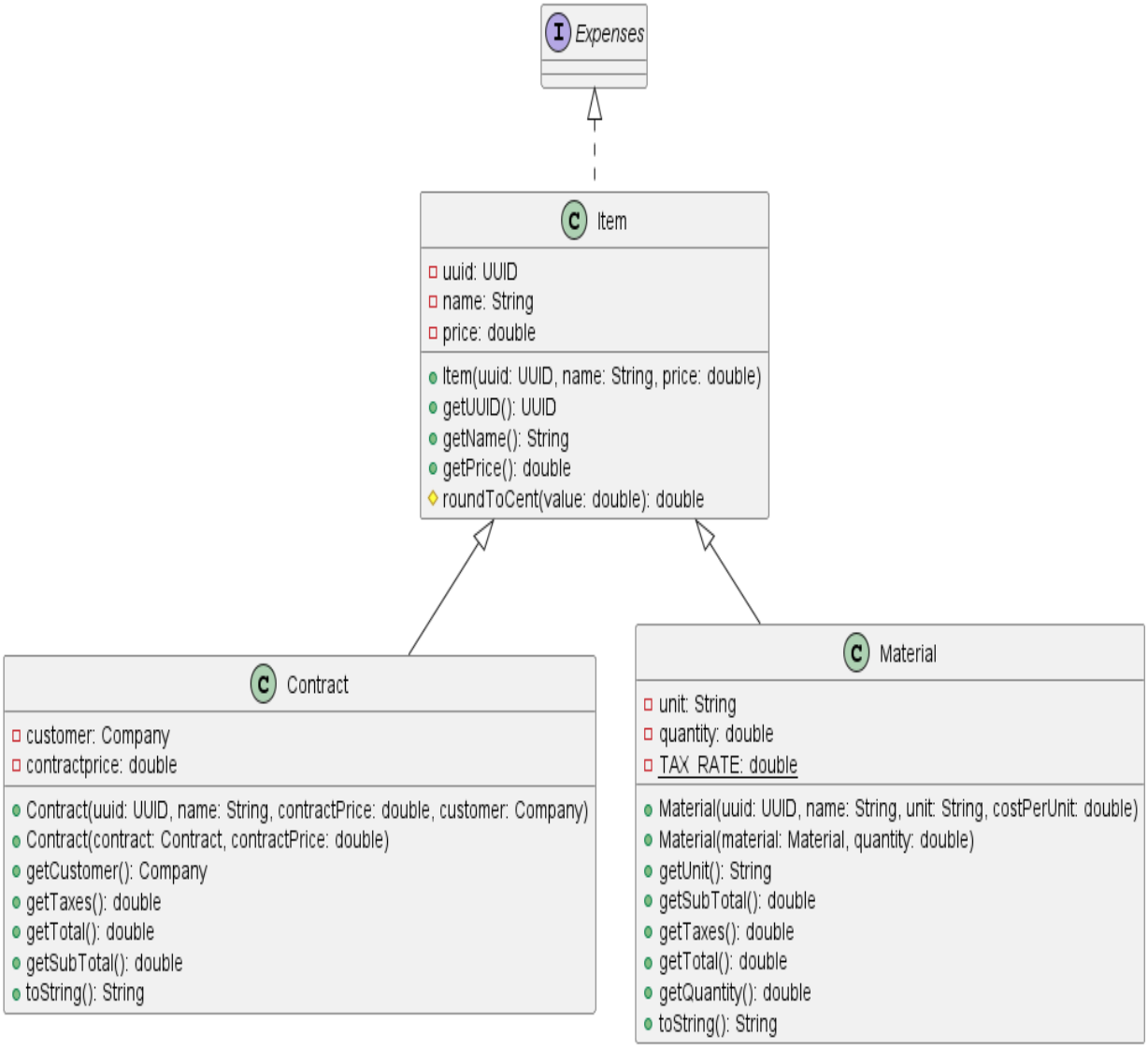

Figure 2.4: Class Structure

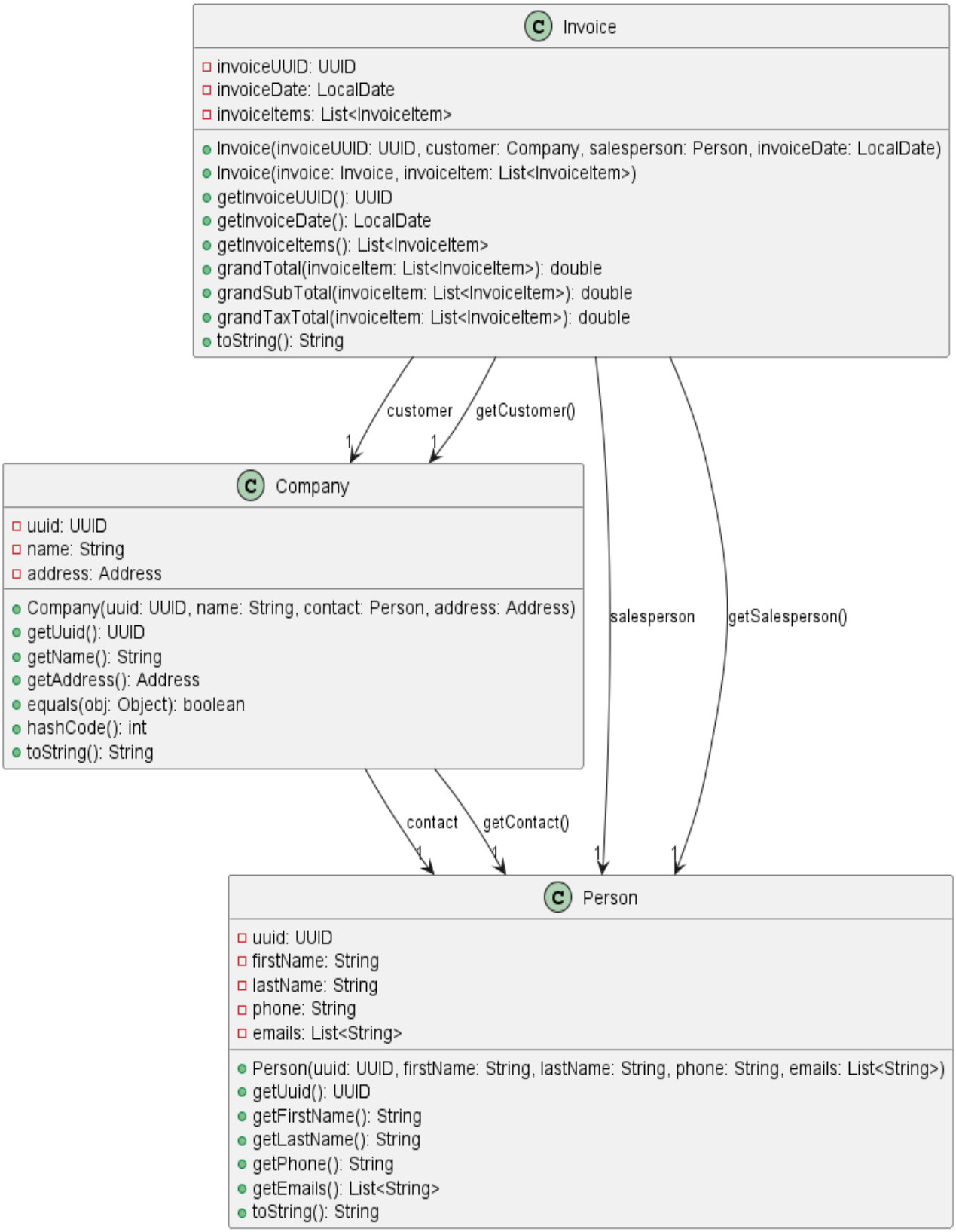

Figure 2.5: Class Structure

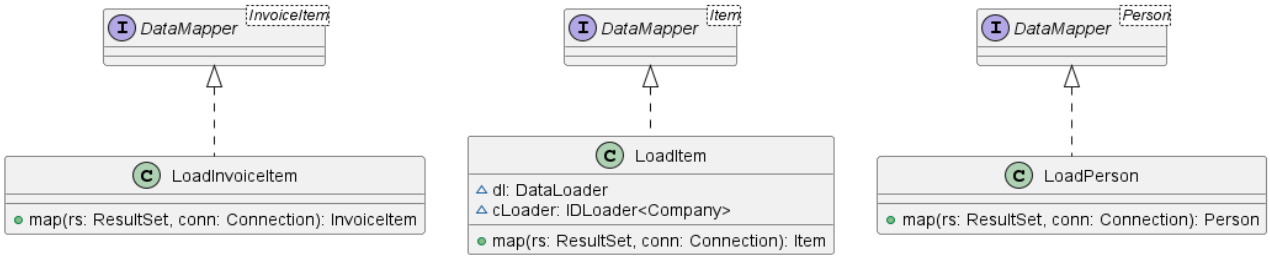

Figure 2.6: Class Structure

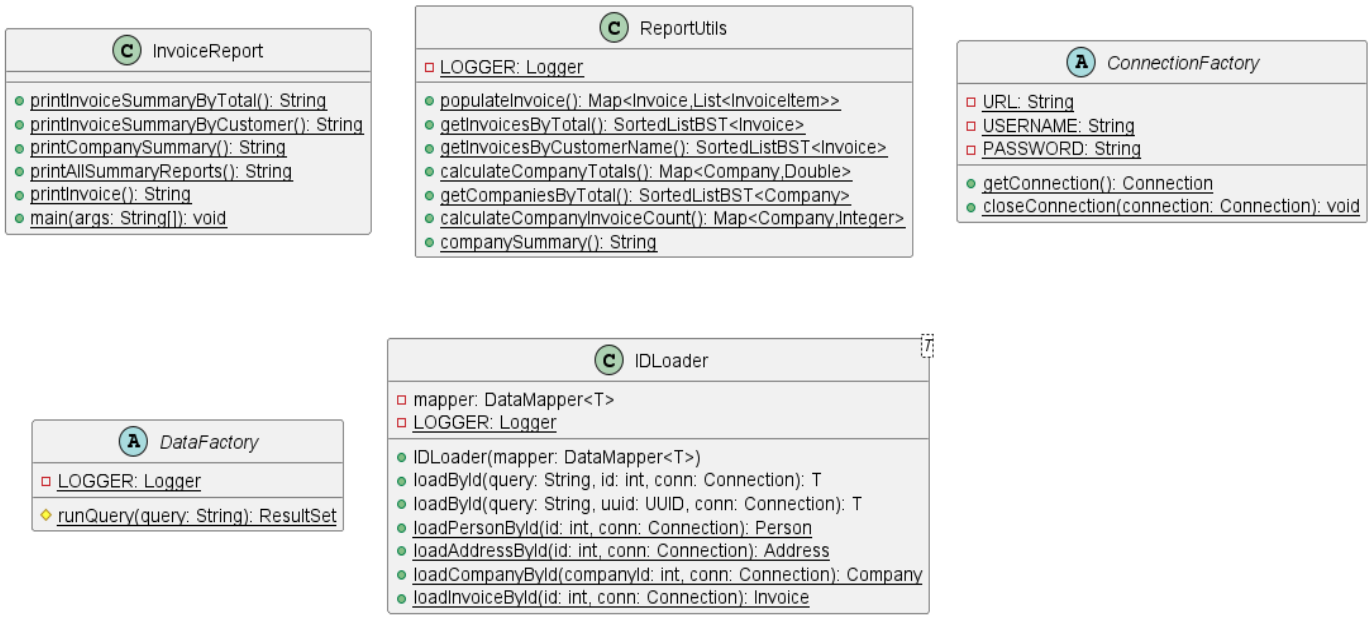

Figure 2.7: Class Structure

3.2.1 Component Testing Strategy

To ensure good design and verify the correctness of implementations, two JUnit test suites are used. The first suite, Entity Tests, includes one test method for each type of item (Equipment, Material, Rental, Lease, and Contract). These tests ensure the correct properties and behavior for each item type. The second suite, InvoiceTests, tests invoices with two items in one case and three items in another. These invoices represent all five item types, verifying that the total calculations for invoices are correct across different item combinations.

3.3 Database Interface

1. Data Loading from Database

   The application interacts with the MySQL database using JDBC and retrieves the data using SQL queries. This is facilitated primarily through the **DataFactory** and **DataLoader** classes.

   - Data Factory methods contain SQL queries that fetch rows from relevant tables and return result sets
   - **Data Loader** orchestrates the loading process by:
     - Executing queries and creating respective Java Objects using DataFactory

2. JDBC-Based Data Management
   - The InvoiceData class provides a set of methods that conform to a defined API for manipulating database records.
   - These methods allow for inserting and removing data from the database.

3. Connection Management
   - Centralized utility class for managing database connections
   - Establish connections using JDBC
   - Ensures safe closure of database resources to prevent leaks
   - Incorporates Log4j for logging connection attempts and errors

4. Corner Cases & Error Handling
   - Null Data Handling: The API ensures that null values do not cause system failures by using checks before processing, such as ensuring non-null data is processed and validated before insertion.

3.3.1 Component Testing Strategy

Corner testing queries are used to detect fraud, and duplicate records are inserted to check integrity. Queries for checking null values and constraints were used to ensure the 1NF, 2NF, 3NF. Junit test is used to test record insertions for database via JDBC.

3.4 Design & Integration of a Sorted List Data Structure

This section details a custom sorted list ADT implemented using Binary Search Tree (BST). The sorted list is implemented using a node-based binary search tree structure. Each node in the tree contains an element and references to the left and right child nodes.

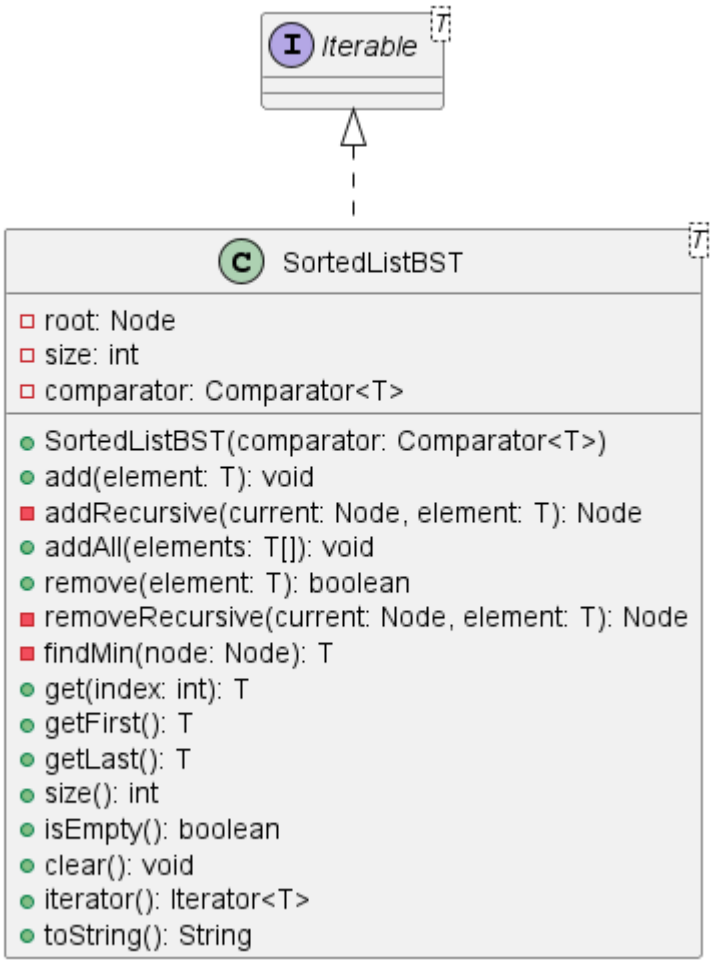

Figure 3: BST Class Structure

Public interface: My SortedListBST<T> class implements the following public interface:

- Constructor: SortedLIstBST (Comparator<T> comparator) - Creates an empty list with the specified ordering
- void add(T element) - Adds an element to the list in its sorted position
- T get(int index) - Returns the element at the specified position in the sorted order
- int size() - Returns the number of elements in the list
- Iterator<T> iterator () - Returns an iterator over the elements in sorted order

The implementation is fully generic by using type parameters (<T>), allowing it to hold any type of object.

- It can store Invoice objects for sorting by total amount or customer name
- It can store Customer objects for sorting by total invoice amount

The application creates three separate sorted lists with appropriate comparators, populates these lists with data retrieved from the MySQL database and uses the natural ordering maintained by the lists to generate the required reports. This integration preserves the existing functionality while using the new sorted list data structure's capabilities.

3.4.1 Component Testing Strategy

Test Scope

Testing focuses on validating the SortedListBST<T> class that implements Iterable<T>, ensuring it maintains correct ordering of elements according to the provided Comparator throughout all operations.

Test Categories

1. Initialization Tests
    a. Verify empty tree initialization
    b. Test construction with different comparators (for total amount, customer name, etc.)
2. Structural Operation Tests
    a. Verify elements are properly ordered when added (highest-to-lowest, alphabetically)
    b. Confirm order is maintained after removals
    c. Test tie-breaking using UUID
3. Core Operations Tests
    a. Add operations (single elements and batch adds)
    b. Remove operations (various positions in the tree)
    c. Iteration functionality (for generating reports)

4. Changes & Refactoring

Using an abstract class (Item) ensures that Material, Contract, and Equipment share common behavior while still implementing Expenses. It prevents code duplication by allowing shared methods (like getTotal()) while forcing subclasses to define specific behavior (like getTaxes()). This improves code reusability, structure, and flexibility. Emails are read into a List which is a String previously. Changing emails from a String to a List allows storing multiple email addresses instead of treating emails as a single piece of text, improving data structure suitability. The original design, where the Company class contains

address attributes directly, is problematic because it violates the Single Responsibility Principle (SRP). The Company class is responsible for both managing company-related data and handling address details, which are separate concerns. By creating a separate Address class, the design becomes more modular, reusable, and easier to maintain.

The Serialization to JSON and XML files is replaced by a MariaDB to manage records and JDBC API to establish connection and execute queries.

5. Additional Material

For logging and exception handling, Log4j is used to capture errors. In the implementation, critical operations are wrapped in try-catch blocks where exceptions are caught and logged with appropriate severity levels. Besides, the SortedListBST implementation utilizes the Comparator interface to maintain elements in sorted order. This design choice allows for flexible sorting by:

- Accepting a custom Comparator in the constructor (SortedListBST(Comparator<T> comparator))
- Using this comparator throughout all operations to determine element placement
- Supporting any type T with a valid comparison logic

Bibliography

[1] Mockaroo, “Random Test Data Generator,” Accessed: Feb. 13, 2025. [Online]. Available: http://www.mockaroo.com