Very Good Building & Development Company

Invoice Subsystem

Yin Po Po Aung

[yaung3@huskers.unl.edu](mailto:yaung3@huskers.unl.edu)

Rometh Samarasinghe  
[rsamarasinghe2@huskers.unl.edu](mailto:rsamarasinghe2@huskers.unl.edu)

University of Nebraska—Lincoln

Spring 2025  
Version 3.0

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 |

Table of Contents

[Revision History 2](#_Toc151733541)

[1. Introduction 4](#_Toc151733542)

[1.1 Purpose of this Document 4](#_Toc151733543)

[1.2 Scope of the Project 4](#_Toc151733544)

[1.3 Definitions, Acronyms, Abbreviations 4](#_Toc151733545)

[1.3.1 Definitions 4](#_Toc151733546)

[1.3.2 Abbreviations & Acronyms 4](#_Toc151733547)

[2. Overall Design Description 5](#_Toc151733548)

[2.1 Alternative Design Options 5](#_Toc151733549)

[3. Detailed Component Description 5](#_Toc151733550)

[3.1 Database Design 5](#_Toc151733551)

[3.1.1 Component Testing Strategy 5](#_Toc151733552)

[3.2 Class/Entity Model 6](#_Toc151733553)

[3.2.1 Component Testing Strategy 6](#_Toc151733554)

[3.3 Database Interface 6](#_Toc151733555)

[3.3.1 Component Testing Strategy 7](#_Toc151733556)

[3.4 Design & Integration of a Sorted List Data Structure 7](#_Toc151733557)

[3.4.1 Component Testing Strategy 7](#_Toc151733558)

[4. Changes & Refactoring 8](#_Toc151733559)

[5. Additional Material 8](#_Toc151733560)

[Bibliography 9](#_Toc151733561)

# 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 both JSON and XML formats.

The report system is designed to generate three types of reports: a summary of all invoices with totals, a summary of invoices grouped by customer, 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.

This project does not hold responsibility for delivery, sales or inventory management system functionalities. The core functionalities of the subsystem include invoice generation, data storage, and structured reporting.

At first, this Subsystem is focused on designing and implementing a Java-based invoice subsystem that can parse input data from CSV files, create object representations of persons, companies, and items and serialize these objects into JSON/XML formats for interoperability. Furthermore, 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

Furthermore, extending functionality and producing reports, this will output 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**: VGB is Very Good Building and Development Company.

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

**CSV:** Comma-Separated Values is a text file that has a specific format which allows data to be saved in a table structured format.

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

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

# 2. Overall Design Description

The Invoice Subsystem is designed to handle data processing, including CSV file conversion to JSON and XML formats. The system is built around several key components: 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.

The subsystem processes various data formats using CSV parsers, 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. The Transaction class provides an abstract blueprint for all transaction types, ensuring that core financial and contractual functionality is implemented in subclasses for Equipment, Material, and Contract items. 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 and multi-item invoice totals.

Following that, the system is extended to retrieve and manipulate data using a MySQL database, replacing the previous flat file-based data loading mechanism. The core business logic, including the driver class, remains largely unchanged. A new database interface layer is 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 serves as the primary API for manipulating database records, providing methods for inserting and deleting data. Additionally, the ConnectionDatabase class manages secure creation and closure of database connections.

## 2.1 Alternative Design Options

It was initially considered introducing two additional classes to distribute the functionalities of the Data Converter class, one dedicated to managing CSV file input and JSON output, and another for handling XML data output. However, given that both functions rely on a single input source, we opted to consolidate these responsibilities within a single class.

For data model structure, the chosen approach is to have separate classes for each entity type rather than using a generic entity class with a type field. This decision ensures type safety and clear separation of concerns, making the system easier to maintain and extend.

For file processing, a custom CSV parser with validation is preferred over third-party CSV libraries. This choice allows for better control over error handling and validation, ensuring that the data being processed meets the required standards and integrity constraints.

For serialization, the system will utilize XStream and Gson libraries instead of manually generating XML and JSON formats. Using well-tested libraries ensures robust and efficient data conversion while reducing development time and minimizing the risk of errors.

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 was considered here but due to added complications to inheritance with Equipment, and the excess implementation it would be facing if Lease and Rental were interfaces, It was decided to have them as Objects and then used by Equipment class.

# 3. Detailed Component Description

This section provides an overview of the database design and its 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. Address Relationships

Geographical Normalization:

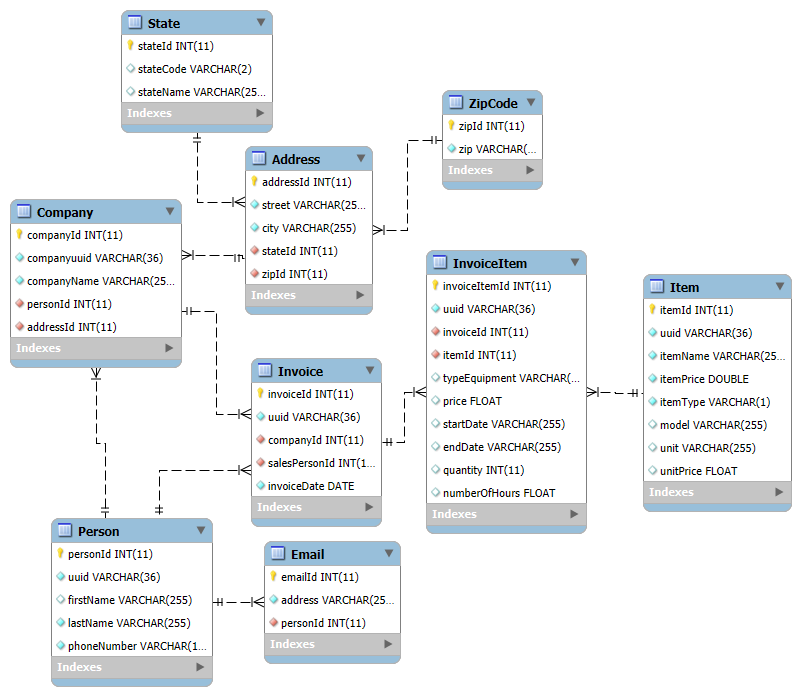
The design follows the third normal form (3NF) principles to reduce redundancy.

* 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

5. 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)

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

 Figure 1: Database Design

### 3.1.1 Component Testing Strategy

## 3.2 Class/Entity Model

The system will convert .csv files and output JSON and XML data objects, This 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

File Processing:

* + CSV parsers for each data type
  + Data validation mechanisms
  + Error handling for malformed data

Serialization Services

* + XML output generation using XStream
  + JSON output generation using Google-Gson
  + Consistent formatting and validation

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

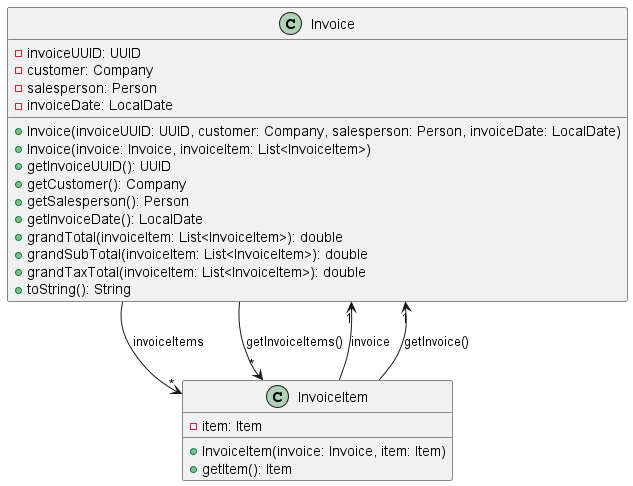
* + EntityTests for validating individual entity calculations
  + InvoiceTests for validating multi-item invoice calculations
  + Test cases covering all five item types and various transaction scenarios

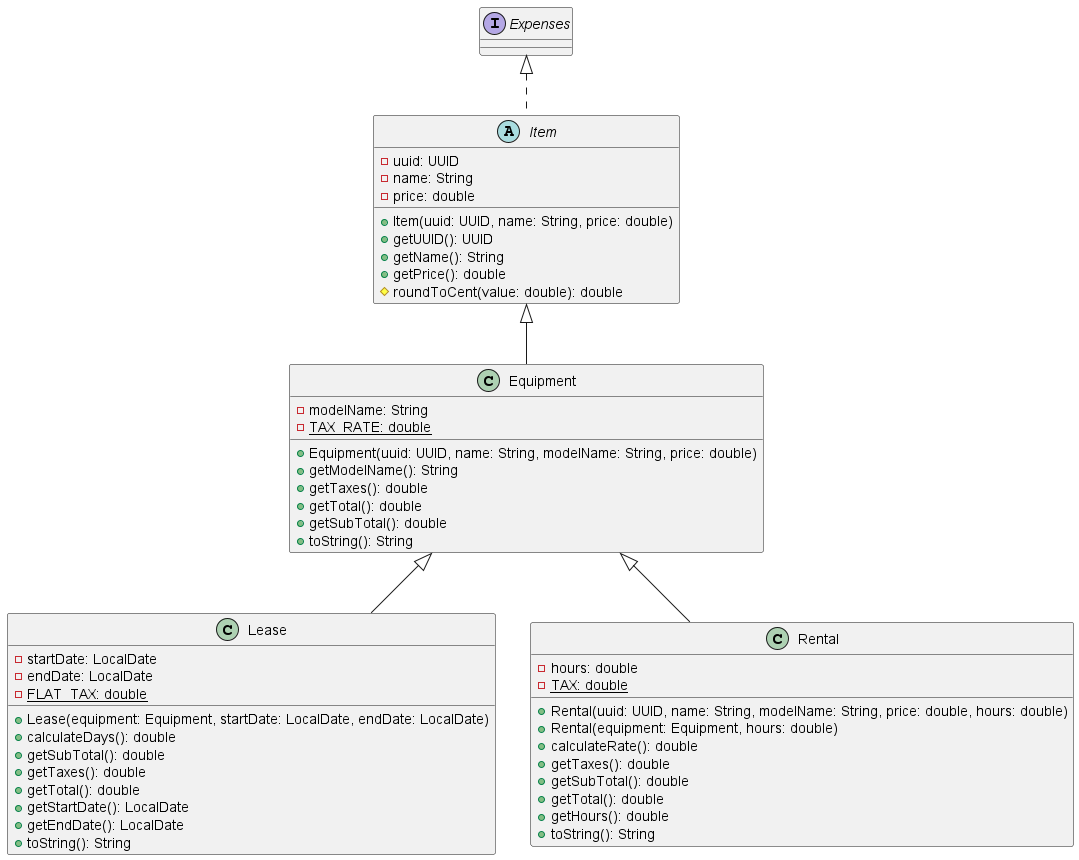
The project branches out to report printing functionalities utilizing the polymorphic behaviors implemented, this functionality will be encapsulated into InvoiceReport class. It will use 3 methods to format 3 different reports and write the report to a file, and one method to output to a text file.

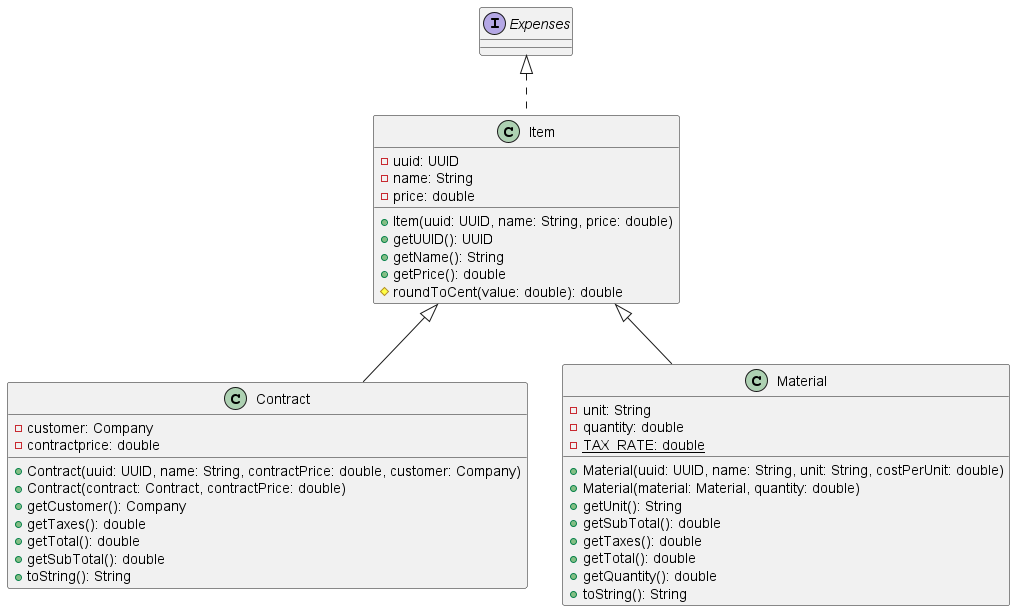
Below is a breakdown of the key components in the diagram:

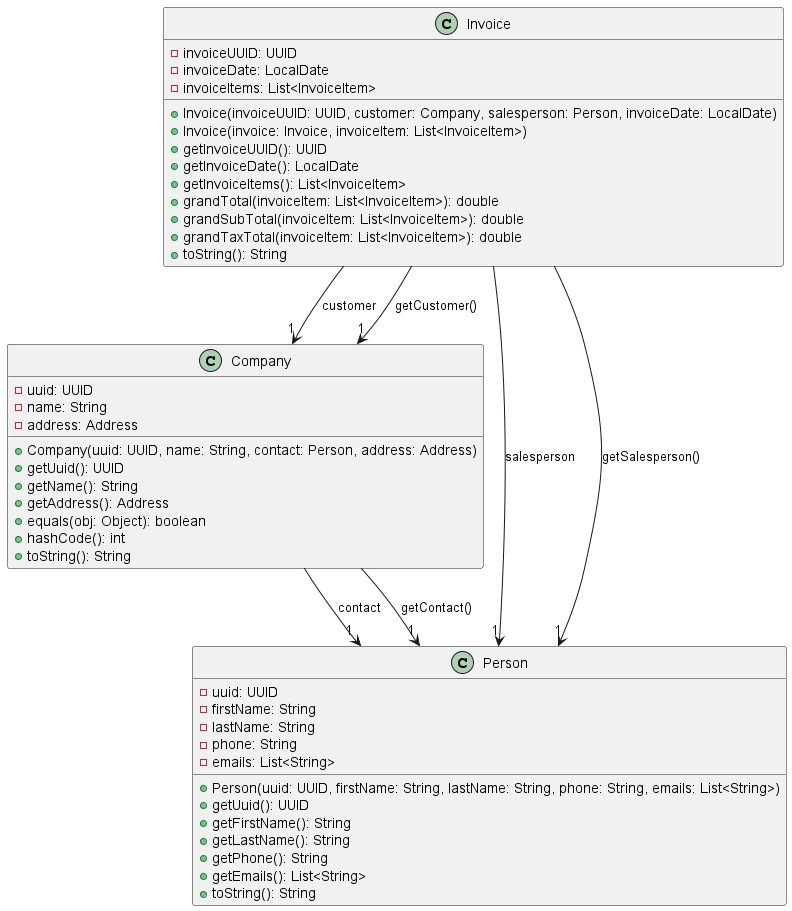


Figure 2.1 : Class Structure

Figure 2.2 : Class Structure

Figure 2.3: Class Structure

Figure 2.4: Class Structure

Figure 2.5: Class Structure

### 3.2.1 Component Testing Strategy

The test data were generated using the following third-party services:

* CSV Generation: Mockaroo[1]
* Data Conversion: ConvertCSV[2]

A total of 9 test cases were created with 3 each for CSV, JSON and XML files, for invoice and invoice item, two csv test cses are created as well.

The following services are used to validate test data.

* JSON Validation: Curious Concept [3]
* XML Validation: W3Schools [4]

To test the inheritance and polymorphic behaviors of the system, JUnit test per functional class were used.

CSV, JSON and XML files are used for testing. JUnit is also integrated to make sure that the program is working. The user testing will be done in an external testing environment.

To ensure good design and verify the correctness of implementations, three 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, will test invoices with two items in one case and three items in another. These invoices will 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 no longer reads from flat files. Instead, it interacts with the MySQL database using JDBC and retrieves the data using SQL queries. This is facilitated primarily through the **DataFactory** and **DataLoader** classes.

* DataFactory methods contain SQL queries that fetch rows from relevant tables and instantiate Java objects using the result sets
* **DataLoader** orchestrates the loading process by:

[This section will be used to detail phase IV where you modify your application to read from a database rather than from flat files. This section will detail the API that you designed–how it conformed to the requirements, how it worked, other tools or methods that you designed to assist, how it handles corner cases and the expectations or restrictions that you’ve placed on the user of the API. What is “good” data and what is considered “bad” data and how does your API handle it? An example table is presented as Table [1](#table:assignmentPerformance).]

Table 1: Average Performance on Assignments; on-time vs. late and individual vs partners. In general, captions for Tables should appear above the table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| On-time | 93.16% (78.46%) | 88.06% (72.31%) | 87.89% (67.69%) | 89.37% (56.92%) | 83.42% (29.23%) | 88.40%  (53.85%) | 74.56%  (75.38%) |
| Late | 88.75% (12.31%) | 85.28% (20.00%) | 70.32% (15.38%) | 90.40% (15.38%) | 82.74% (44.62%) | 94.22%  (15.38%) | N/A |
| Diff | 4.42% | 2.79% | **17.57%** | 1.03% | 0.68% | 5.82% | - |
| Individual | NA | 88.43% (73.85%) | 82.32% (33.85%) | 87.22% (27.69%) | 86.40% (23.08%) | 82.67% (26.15%) |  |
| Pairs | NA | 83.55% (18.46%) | 86.22% (49.23%) | 91.00% (46.15%) | 78.53% (49.23%) | 92.83%  (46.15%) |  |
| Diff | NA | 4.88% | 3.90% | 3.78% | 7.87% | 10.16% |  |

### 3.3.1 Component Testing Strategy

[This section will describe your approach to testing this particular component. Describe any test cases, unit tests, or other testing components or artifacts that you developed for this component. How was test data generated (if a tool was used, this is a good opportunity for a citation). How many test cases did you have; how many of each type? *Justify* why that is sufficient. What were the outcomes of the tests? Did the outcomes affect development or force a redesign?

You may refer to the course grader system as an external testing environment “provided by the client” or “another QA/testing team”.]

## 3.4 Design & Integration of a Sorted List Data Structure

[This section will be used to detail phase V where you design and implement a custom data structure and integrate it into your application. Is your list node based or array based? What is its *interface* and how does it define a sorted list? Is it generic? Why? You can/should provide another UML diagram for this list.]

### 3.4.1 Component Testing Strategy

[This section will describe your approach to testing this particular component. Describe any test cases, unit tests, or other testing components or artifacts that you developed for this component. How was test data generated (if a tool was used, this is a good opportunity for a citation). How many test cases did you have; how many of each type? *Justify* why that is sufficient. What were the outcomes of the tests? Did the outcomes affect development or force a redesign?

You may refer to the course grader system as an external testing environment “provided by the client” or “another QA/testing team”.]

# 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 were read into a List which was 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 contained address attributes directly, was problematic because it violated the Single Responsibility Principle (SRP). The Company class was responsible for both managing company-related data and handling address details, which are separate concerns. By creating a separate Address class, the design became more modular, reusable, and easier to maintain.

# 5. Additional Material

[This is an optional section in which you may place other materials that do not necessarily fit within the organization of the other sections.]

# Bibliography

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

[2] ConvertCSV, “Online CSV Conversion and Data Generator,” Accessed: Feb. 13, 2025. [Online]. Available: <https://www.convertcsv.com>

[3] Curious Concept, “JSON Formatter & Validator,” Accessed: Feb. 13, 2025. [Online]. Available: <https://jsonformatter.curiousconcept.com/>

[4] W3Schools, “XML Validator,” Accessed: Feb. 13, 2025. [Online]. Available: <http://www.w3schools.com/xml/xml_validator.asp>