

CCC Invoicing Subsystem

Minh Tran

mtran33@huskers.unl.edu

University of Nebraska—Lincoln

Spring 2026

v1.5

This document is a technical design specification for the CCC Invoicing Subsystem, an object-oriented application designed to modernize billing and business operations for Cinco Computer Consultants.

Revision History

Version	Change(s) description	Author(s)	Date
1.0	Document created	Minh Tran	2026/02/02
1.1	Detailed filled in up to Section 3.2	Minh Tran	2026/02/03
1.2	First draft update	Minh Tran	2026/02/06
1.3	Added some extra details	Minh Tran	2026/02/10
1.3.1	Output formats added	Minh Tran	2026/02/13
1.4	Second draft update	Minh Tran	2026/02/20
1.5	Assignment 2.0 revision	Minh Tran	2026/02/27

Contents

1	Introduction	3
1.1	Purpose of this Document	3
1.2	Scope of the Project	4
1.3	Definitions, Acronyms, Abbreviations	4
1.3.1	Definitions	4
1.3.2	Abbreviations & Acronyms	4
2	Overall Design Description	4
2.1	Alternative Design Options	5
3	Detailed Component Description	5
3.1	Database Design	5
3.1.1	Component Testing Strategy	5
3.2	Class Model	6
3.2.1	Component Testing Strategy	7
3.3	Database Interface	7
3.3.1	Component Testing Strategy	7
3.3.2	Component Testing Strategy	8
3.4	Design & Integration of a Sorted List Data Structure	8
3.4.1	Component Testing Strategy	8
4	Changes & Refactoring	9
5	Additional Material	9
6	Notes	9
	Bibliography	9

1 Introduction

This document serves as the design specification for Cinco Computer Consultants (CCC), a regional company specializing in computer equipments, services, and licenses. The project is focused on modernizing CCC's business operations by replacing a legacy system of ad-hoc Excel sheets and MS Access databases with a sophisticated, integrated invoicing subsystem.

The CEO has proposed item price requirements as below:

- **Equipment:** Physical hardware sold or leased. If sold, calculates with the formula $\text{pricePerUnit} \times \text{amount}$. If leased, calculates with the amortization formula: $\frac{\text{pricePerUnit} + \text{pricePerUnit} \times 0.5}{36}$.
- **Service:** Professional labor such as training and installation. These are billed per hour, plus a flat service fee: $\text{hours} \times \text{rate} + \text{serviceFee}$.
- **License:** Software or third-party services. These are billed based on a specific duration, plus an annual fee: $\frac{\text{days} \times \text{annualServiceFee}}{365} + \text{flatServiceFee}$.

The client's business model relies on a B2B (business-to-business) approach. Key features include:

- **Inventory management:** Categorizing items into sections with unique data requirements: Equipment, Services, and Licenses.
- **Financial logic:** Implements automated calculations: Equipment markups, lease amortization, and variable service fees. Using polymorphism, it ensures accurate billing by applying specific business rules to calculate fees.
- **Taxation system:** Implements different tax rates (use tax, sale tax) based on item type and price thresholds.
- **Reporting system:** After all calculations are done, it generates a comprehensive summary in both XML and JSON formats.

1.1 Purpose of this Document

The purpose of this document is to provide a technical blueprint for the invoicing subsystem. It outlines the architectural design, data models, and business logic required to satisfy the requirements set by the CEO Steve Brule. It serves as a guide for developers and technical audiences to ensure the final implementation aligns with the company's operational needs.

1.2 Scope of the Project

The project covers the development of a standalone invoice subsystem.

- **Included:** Data persistence, billing formulas implementation, formatted invoice reports generation.
- **Not included:** Marketing, delivery, general inventory management, future recurring billing.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Amortization: The process of spreading the cost of a marked-up equipment lease over a fixed 3-year period (36 months).

1.3.2 Abbreviations & Acronyms

CCC Cinco Computer Consultants

UUID Universally Unique Identifier

B2B Business-to-Business

OOP Object-oriented programming

EDI Electronic Data Interchange

CSV Comma-Separated Values

XML eXtensive Markup Language

JSON JavaScript Object Notation

2 Overall Design Description

The invoicing system is designed using an object-oriented architecture. It relies heavily on polymorphism and inheritance to handle different billable entities without writing redundant, procedural code. The architecture is broadly divided into three primary responsibilities:

- **Data input:** The **Input** class acts as a parser, reading in flat files (.csv) and initializing base entity objects.

- **Processing:** The core of the system is built around a robust class hierarchy. An `Invoice` object aggregates various `Item` objects. By defining a common contract in the abstract `Item` class, the system can calculate taxes, totals, and fees polymorphically by calling overridden methods in the `Equipment`, `Service`, and `License` subclasses.
- **Data output:** The `Output` class serializes the processed in-memory objects into structured XML and JSON formats for external system integration.

2.1 Alternative Design Options

One of the other alternatives of object-oriented programming would be to utilize `if/else` statements to determine billing math based on item types. However, this alternative was quickly discontinued because it has proved to be extremely difficult to maintain in the long run, if CCC decides to slightly change the calculation logic or add a new billable item type in the future.

3 Detailed Component Description

3.1 Database Design

In this project, MySQL will be used as the main database interface.

3.1.1 Component Testing Strategy

The integrity of the database can be tested using a suite of SQL testing files, specifically designed to attempt "illegal" operations.

So far, these methods are the "debuggers" for the program:

- **Invalid UUID method:** This method will try to create an invoice for a non-existent customer UUID to verify the database rejects the transaction.
- **Invalid type method:** This method will try to insert strings into a number column, such as the `costPerItem` column or the `costPerHour` column to ensure proper data types are maintained.
- **Invalid number of fields / null fields:** This method will try to create an invoice, but with blank data fields (such as a `UUID` field having a `null` value).

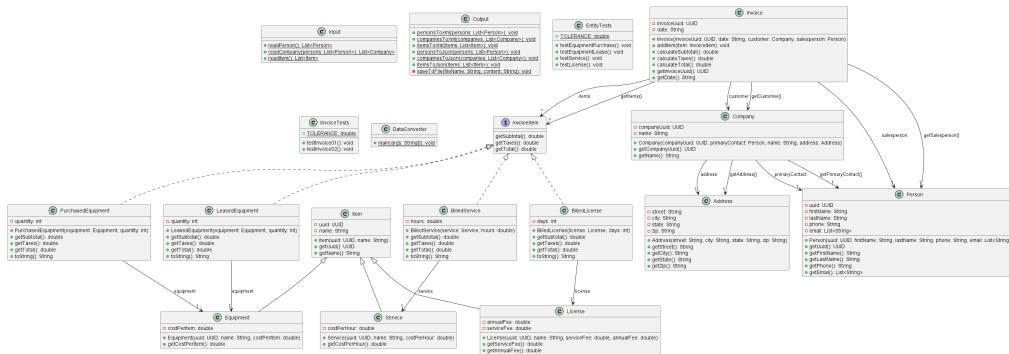


Figure 1: A PlantUML overview of the project.

3.2 Class Model

Main class files:

- **DataConverter:** Has a main method and calls other methods from the files listed below.
- **Input:** A class designed to parse .csv files. This program focuses on files named `Companies.csv`, `Persons.csv`, and `Items.csv` located inside the `/data` folder.
- **Output:** An exporter class, which exports data gotten from `Input.java` into detailed XML and JSON files in the `/data` folder, using Xstream and Gson packages.

Definition class files:

- **Item:** Has `uuid` (UUID), `name` (String), with methods `getTaxes()` (double) and `getTotalPrice()` (double) which requires implementation from the subclasses. It is an abstract class and represents an item below:
 1. **Equipment:** Represents an equipment. Has a `costPerItem` (double).
 2. **Service:** Represents a service. Has a `costPerHour` (double).
 3. **License:** Represents a license. Has a `serviceFee` (double) and a `annualFee` (double).
- **Person:** A class that either represents a salesperson or a representative primary contact of a company. Has a `uuid` (UUID), `firstName` (String) and `lastName`, a `phone` (String), and `email` (List<String>).
- **Company:** A class that represents a company/customer. Has a `companyUuid` (UUID), a `primaryContact` (Person) and an `address` (Address).
 1. **Address:** Groups data fields `street`, `city`, `state`, `zip` into one constructor.
- **Invoice:** A container class that holds a List<Item>. It also includes methods such as `calculateTotal()` and `calculateTaxes()`, works by iterating over its

list of items.

JUnit testing class files:

- **EntityTests**
- **InvoiceTests**

3.2.1 Component Testing Strategy

- **Unit tests:** Individual tests is written for each subclass to verify their math. For example, a test case for **Service** asserts that 10 hours at \$50/hr plus a \$150 fee equals exactly \$650.00.
- **Rounding tests:** To adhere to the "Round to nearest cent" rule, edge cases (for example: values ending in .005) is added to ensure consistent rounding behavior.

3.3 Database Interface

The database interface layer serves as the bridge between the Java application and the SQL persistence store. This component is implemented using the Java Database Connectivity (JDBC) API. A singleton **DatabaseManager** class is responsible for establishing connections, managing transactions, and executing pre-compiled SQL statements.

Data Validation & Safety:

- **SQL Injection Prevention:** All database queries utilize **PreparedStatement** objects. This ensures that "bad data" (such as a customer name containing SQL commands like `"; DROP TABLE;"`) is treated as a literal string rather than executable code.
- **Referential Integrity:** The interface checks for the existence of Foreign Keys. If an invoice references a Customer UUID that does not exist in the database, the API throws a custom **DataIntegrityException** rather than processing the invalid record.

3.3.1 Component Testing Strategy

Testing the database interface involved integration tests against a live test database (separate from production data). We utilized a suite of 20 test cases, split between:

- **Happy Path:** Valid data insertion and retrieval. Verifying that a saved Invoice comes back with the exact same items and totals.
- **Destructive Testing:** deliberately inserting invalid foreign keys or null values into required fields to verify that the Java **SQLException** is caught and handled

gracefully by the logging system.

The testing revealed a bottleneck in connection opening/closing, which forced a redesign to implement a connection pooling strategy (keeping the connection open for batch operations).

3.3.2 Component Testing Strategy

3.4 Design & Integration of a Sorted List Data Structure

```
/**
 * A basic Hello World class
 */
public class HelloWorld {

    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

Figure 2: A basic Hello World program in Java.

If you do use the minted package, you may need a substantial amount of additional setup. First, you need to install the L^AT_EX `minted` package installed *and* you need to install the python `pygments` package (see <https://pygments.org/download/>; you can use `pip install pygments`). Once everything is installed, you must compile using the `--shell-escape` flag.]

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

5 Additional Material

6 Notes

References