CCC Invoicing Subsystem

Minh Tran
mtran33@huskers.unl.edu
University of Nebraska—Lincoln

Spring 2026
v1.3.1

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

Contents

1 Introduction

This document serves as the design specification for Cinco Computer Consultants (CCC), a regional company specializing in computer services including training, consultations, 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.

This project is a Java-based, object-oriented application designed to keep track of client invoices. The system tracks a variety of billable items - such as hardware, leases, and professional consulting hours. It ensures accurate billing by applying specific business rules for markups, service fees, and taxation.

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.
- **Taxation system:** Implements different tax rates (use tax, sale tax) based on item type and price thresholds.
- **Reporting system:** After all calculations are done, 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 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

Equipment: Physical hardware sold or leased. These are billed based on whether it is sold or leased.

Service: Professional labor such as training and installation. These are billed per hour, plus a flat service fee.

License: Software or third-party services. These are billed based on a specific duration, plus an annual fee.

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

XML eXtensive Markup Language

JSON JavaScript Object Notation

2 Overall Design Description

The invoicing system is designed using an object-oriented architecture in Java. Inside, the system utilizes a polymorphic approach to handle different invoice items, allowing the central billing engine to process equipments, services, and licenses through a common interface while respecting their unique calculation rules.

2.1 Alternative Design Options

As of the last document update, there has not been an alternative design option. Object-orientation is the current design option.

3 Detailed Component Description

3.1 Database Design

The database schema is designed to enforce data integrity and minimize redundancy. To meet the requirement that every item is identified by a UUID, the primary keys for all major entities are stored as 32-character strings (rather than auto-incrementing integers).

The schema consists of these tables:

1. **Persons:** Stores individual people's contact data. It is used for both customer contacts and salespersons.
2. **Customers:** Stores company data. It also stores a foreign key that links to a primary contact in the Persons table.
3. **Products:** A polymorphic table storing defining attributes for Equipment, Licenses, and Services. A `type` column acts as a discriminator (in this case, 'E', 'L' and 'S') to indicate which subsequent columns (like `hourly_rate` vs `unit_price`) are relevant.
4. **Invoices:** The central entity linking a Customer and a Salesperson to a specific transaction date.
5. **InvoiceItems:** A relational join-table that links a specific Invoice to a Product. It records transaction-specific data, such as the number of hours billed for a service or the specific lease term for equipment.

3.1.1 Component Testing Strategy

The integrity of the database can be tested using a suite of SQL scripts 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 text into a number column, such as the `unit_price` column to ensure proper data types are maintained.

3.2 Class/Entity Model

The Java class structure mirrors the database schema but adds behavior through methods. The core logic is encapsulated in the `Item` class hierarchy.

- **Invoice class:** 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.
- **Item (abstract class):** Defines the contract for all billable entities. It requires subclasses to implement `getTaxes()` and `getTotalPrice()`.
- **Concrete Subclasses:**
  - `Equipment`: Handles logic for purchase vs. lease. If leased, calculates with the amortization formula: $\frac{\texttt{Cost}+\texttt{Cost}\times 0.5}{36}$.
  - `License`: Handles the dates to calculate the cost based on the number of days effective. Calculates cost based on $\frac{\texttt{days}\times\texttt{annualServiceFee}}{365} + \texttt{flatServiceFee}$.
  - `Service`: Calculates cost based on $\texttt{hours} \times \texttt{rate} + \texttt{serviceFee}$.

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).

Table 1: Average Performance on Assignments; on-time vs. late and individual vs partners. In general, captions for Tables should appear above the table. This is just an example of how to properly include a 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%	

3.3.2 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.

You should not include large chunks of code in your document, but if you do need to typeset code in LaTeX the recommendation is the `minted` package. You can typeset code inline like this: `List<Integer> myList` or you can define entire blocks (usually placed within a figure environment) like in Figure 1.

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

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

Figure 1: 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 LaTeX `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

References