|  |
| --- |
| Department of computer science & Engineering  University of Nebraska—Lincoln |
| Cinco Computer Consultants Invoice System |
| Computer Science II Project |
|  |
| **I Coleman, Sarah Kenny, and Nisha Rao** |
| **3/18/2016**  **Version 4.0** |

|  |
| --- |
| This document contains the description and design of a substantial database-backed object-oriented invoice system in Java. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Description of Change(s) | Author(s) | Date |
| 1.0 | Initial draft of this design document (in preparation for phase 1) | I Coleman, Sarah Kenny, Nisha Rao | 2016/02/02 |
| 2.0 | Second draft in preparation for phase 2 | I Coleman, Sarah Kenny, Nisha Rao | 2016/02/15 |
| 3.0 | Third draft in preparation for phase 3 | I Coleman, Sarah Kenny, Nisha Rao | 2016/03/04 |
| 4.0 | Fourth draft in preparation for phases 4 and 5 | I Coleman, Sarah Kenny, Nisha Rao | 2016/03/18 |

Contents

[Revision History 1](#_Toc349390336)

[1. Introduction 4](#_Toc349390337)

[1.1 Purpose of this Document 4](#_Toc349390338)

[1.2 Scope of the Project 4](#_Toc349390339)

[1.3 Definitions, Acronyms, Abbreviations 4](#_Toc349390340)

[1.3.1 Definitions 4](#_Toc349390341)

[1.3.2 Abbreviations & Acronyms 4](#_Toc349390342)

[2. Overall Design Description 5](#_Toc349390343)

[2.1 Alternative Design Options 5](#_Toc349390344)

[3. Detailed Component Description 5](#_Toc349390345)

[3.1 Database Design 5](#_Toc349390346)

[3.1.1 Component Testing Strategy 7](#_Toc349390347)

[3.2 Class/Entity Model 8](#_Toc349390348)

[3.2.1 Component Testing Strategy 9](#_Toc349390349)

[3.3 Database Interface 10](#_Toc349390350)

[3.3.1 Component Testing Strategy 10](#_Toc349390351)

[3.4 Design & Integration of Data Structures 10](#_Toc349390352)

[3.4.1 Component Testing Strategy 10](#_Toc349390353)

[3.5 Changes & Refactoring 10](#_Toc349390354)

[4. Bibliography 11](#_Toc349390356)

# Introduction

This is a software design description for the simple invoice system that will replace Cinco Computer Consultants’ aging AS400 green-screen system. This software will be object-oriented, written in Java, and support CCC’s business model by implementing their business rules and providing the functionality they have requested.

## Purpose of this Document

This document is intended to detail the functionality of the new invoice system and communicate the ideas behind its creation. It will cover the scope of the project and the technical design of a class/entity model and database. As this document is updated throughout the project, it will also reflect changes, refactoring, and revisions made to both the document and the software.

## Scope of the Project

This invoice system is being developed for CCC, a company which sells electronic equipment, provides consultation services, and licenses software, server hosting, and third-party services. The system will be backed by a mySQL relational database which will allow the program to update, store, and retrieve CCC’s data in the form of Java-based invoice objects.

## Definitions, Acronyms, Abbreviations

### Definitions

**Electronic Data Interchange:** The transfer of objects between different systems which may use completely different languages and technologies.

**Object-oriented Programming:** a programming model structured around manipulable objects rather than the logic require to manipulate them.

### Abbreviations & Acronyms

**API** – Application Program Interface

**EDI** – Electronic Data Interchange

**ER** – Entity-Relationship (model)

**JDBC** – Java Database Connectivity

**JSON** – JavaScript Object Notation

**OOP –** Object-Oriented Programming

**SQL** – Structured Query Language

**UML** – Unified Modeling Language

**XML –** Extensible Markup Language

# Overall Design Description

The invoice system utilizes three main types of objects: Persons, Customers, and Products. These objects, along with their subclasses, model real-world entities relevant to CCC’s business. Relevant data, methods, and functionality are built into these objects with good encapsulation enforced. The goal of using these objects is to promote reusability and proper abstraction, rather than storing data in a procedural function that must be changed to suit different purposes.

For data storage, the program uses a MySQL database. Each object in the application exists in the database as a table with columns for the object’s specific fields (for example, each Person has a first name, last name, and mailing address, among others).

## Alternative Design Options

Instead of using a database, we could have chosen to store data in a flat data file. This would have been much simpler and easier to execute. However, relational databases allow for data to be shared across networks and offer more robust reporting. They also allow users to perform search queries based on keys or indexes, letting CCC find information far more quickly. Finally, relational database management systems mean we can control the read/write permissions of the data.

In addition, instead of creating an object-oriented system, we could have used procedural programming. Using OOP allowed us to create more complicated behavior with less code, and made it possible to reuse code – adding features to a procedurally-programmed system in the future would have been much more difficult.

# Detailed Component Description

This section describes in detail each individual aspect of the system. There is a description of the database design, the design for the class model system, the database interface, and the implemented data structure.

## Database Design

The design of our mySQL database tables (shown on the ER diagram below) closely mirror our Java class objects. Every class is now reflected in a table: Persons, Customers, Products, etc. Each table contains records that pertain to the object of the same name. However, each table contains an additional element – a unique, auto-generated primary key which is assigned to each data record to preserve data integrity.

These primary keys are also used to link records from different tables. Each entry in the Invoice table must contain a Person and a Customer. This is accomplished by storing one entry’s primary key (for example, Customer) as a foreign key in another table (for example, Invoice). When a table contains a foreign key, all child records or tables must be deleted prior to the deletion of the parent record or table. For example, to delete a Person, one must first delete its Address. Since an entry in Invoice is a parent of every other table, a record must be deleted from every other table before deleting an Invoice.

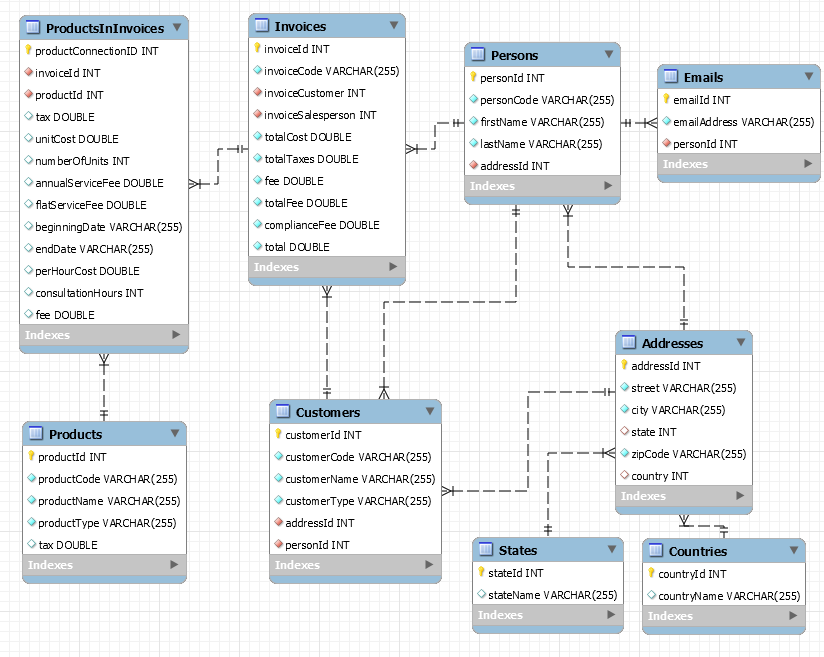
Sometimes, multiple entries in one table must be connected to a single entry in another table. For this, we use connector tables, like the ProductsInInvoices table which contains a foreign key to a Product and a foreign key to an Invoice. Since an Invoice may contain any number of Products, we can’t just use a single foreign key in the Invoices table like we can with Persons or Customers. Instead, we can query ProductsInInvoices for all products matching a particular Invoice’s primary key. A constraint ensures that each Invoice-Product combination is unique.

This is also why we created a separate table for Emails – one person may have any number of E-mail addresses or even no E-mail address at all, so we can query the Emails table to return all addresses associated with a particular personId.

Finally, we also made separate tables for States and Countries in order to normalize geographical data in the Addresses table – that is, to make sure locations in the same state refer to only one state record.

The database schema has the ability to create, retrieve, update, and destroy data as needed while maintaining data integrity.

**Figure 1: Entity-Relation diagram of the database**

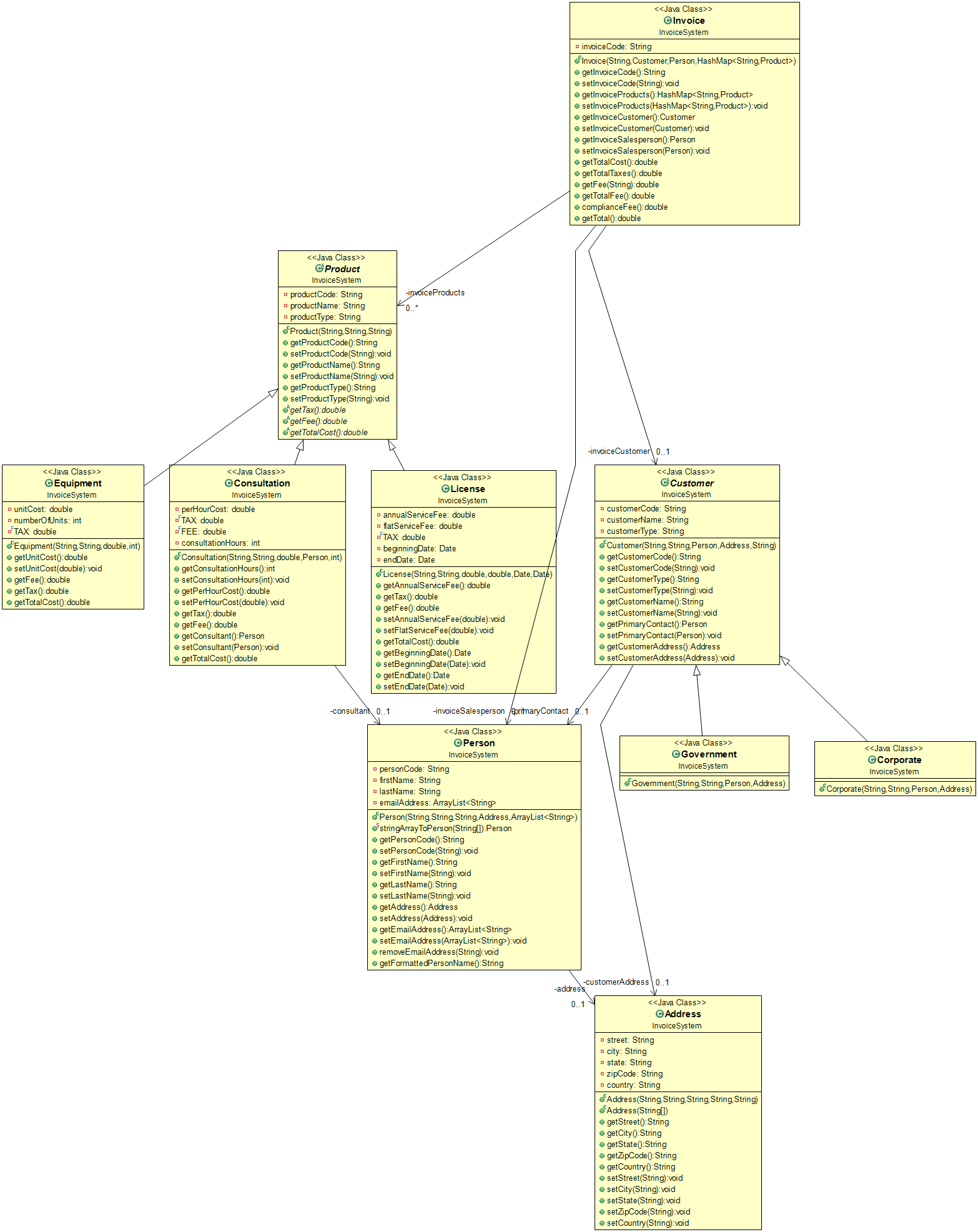


### Component Testing Strategy

Implementation and testing of the database was conducted using MySQL Workbench.

The same data used to test the Java API was used to test the database. It was inserted, deleted, and changed in the tables using various queries. To ensure that all data was being properly entered into the tables, and that data could be easily found within the tables, more queries were used only for the retrieval of data and to make sure that the modification queries executed properly. The goal of these queries was to test every request that could be made of the database.

## Class/Entity Model



**Figure 2: UML diagram of Java classes, generated using [1]**

Our invoice system uses classes to represent instances of specific objects. These classes contain constructor methods to create objects using provided data, data which is encapsulated in the class it is used to create. By default, all member variables of classes are set to private. Variable interfacing is therefore handled by methods contained in each class which “get” values or “set” values.

This application models three main classes: Persons, Customers, and Products. Persons contain a unique alpha-numeric designation from the old system, a name, a mailing address, and an array list of E-mail addresses. Mailing addresses, which contain many parts, are modeled as a separate class called “Address”. The Person class also contains a method used to print the person’s name as a well-formatted string.

Products contain a code, a name, and a type (equipment, license, or consultation).These three types are modeled as different subclasses: Licenses, Equipment, and Consultations. These subclasses inherit all of the Product class’ non-private methods and member fields, but contain their own specialized fields and methods to add more functionality. This models real-world systems, which often contain entities related to each other.

Equipments contain a price per unit, a number of units, and a tax rate. Licenses contain service and annual license fees, a tax rate, and a beginning and end date for the license, modeled using the standard java.util.Date class. Consultations contain a per hour cost, an hourly fee, the number of hours the Consultation took, a tax rate, and a Person object representing a consultant. Each Product subclass also contains methods to calculate their own total cost.

Customers contain a unique alpha-numeric designation from the old system, a primary contact, a name, and a mailing address. Customers can be Corporate or Government subclasses, which are functionally identical. The primary contact is a Person object, and the mailing address is again represented as an Address object.

The Invoice class is the overarching summary class – it contains an alphanumeric code to identify itself, a Customer, a salesperson modeled as a Person, and a hash map of Products. It also contains a number of useful functions for financial reports, such as getTotalCost(), getTotalTaxes(), getTotalFee(), etc. (the complete list of functions is listed in the above UML diagram.) Invoice also checks whether or not the Customer is a Government customer and calculates compliance fees accordingly. These functions use each Product subclasses’ getTotalCost() function

### Component Testing Strategy

We tested our initial class constructors by passing in three loosely-formatted data files containing information pertaining to Customers, Persons, and Products. These data files came from the old system, and we and our colleagues working on similar systems created 25 more test cases formatted the same way. We created a function as part of our class package which read in these test files, put the data therein into the classes we designed, and then output that data as XML and JSON files. These files contain code in their respective languages that presents each individual field in each individual entry, allowing us to visualize our classes. This not only allowed us to see if our classes were working correctly, but demonstrated that our objects were versatile enough to be presented in any format CCC desires.

We tested the Invoice class by giving it a .dat file similar to those used for Customers, Persons, and Products. This was then used to output a summary of all invoices and a detailed report of all classes.

## Database Interface

After the database is constructed in mySQL, the necessary data is entered into the database tables using INSERT INTO queries. Then, a driver class in the Java program loads the database using JDBC. This allows the Java application to connect to the database to retrieve, update, insert, and delete data. Results from retrieval queries are stored in a result set and processed accordingly.

It is important that the Java application, not the database, deals with any problems that may occur during the connection. All the SQL queries the driver class uses are designed to sanitize malicious or unwanted SQL injections. Methods that remove data are structured to ensure data integrity is preserved. Methods that accept data as parameters check to ensure that the data is valid to prevent impossible values from being inserted into tables. This includes checking for null values, which are allowed in some cases (for example, not every Address has a State), but not allowed in others (every Address must have a Country).

### Component Testing Strategy

Similar to section 3.1.1, the database interface was tested by populating the database with information, then deleting, updating, and retrieving tables and specific information from tables. This time, however, this was done not by querying the database directly, but by writing a test application class using JDBC. The tests worked with both valid and invalid data to make sure the latter generated an error message. Our program also generated a summary report to make sure the data was being stored correctly.

## Design & Integration of Data Structures

The system uses a custom API in the InvoiceData class, a generalized service from which a client (web app, GUI application, etc.) can access and alter the data stored in our database. These methods are: removeAllPersons, removePerson, addPerson, addEmail, removeAllCustomers, removeAllProducts, removeProduct, adEquipment, addLicense, addConsultation, removeAllInvoices, removeInvoice, addInvoice, addEquipmentToInvoice, addLicenseToInvoice, and addConsultationToInvoice, all of which do what their names suggest. InvoiceData uses the JDBD interface described in section 3.3 of this document, ensuring that no bad data is passed in.

### Component Testing Strategy

A testing client written by a third party was used to access our API, delete all the information contained in the database, produce summary reports and verify the correct output, all using the InvoiceData class.

## Changes & Refactoring

After the testing and creation of the Invoice class, major refactoring occurred to bring the system more in line with the principles of object-oriented programming (most significantly, encapsulation, the principle of binding all data together with the function that manipulates the data). Around the publication of version 2 of this document, most of the numerical data for Products (fees, taxes, etc.) was contained in a series of hash maps in a main function, or in another class called Invoice Transactions that existed solely to manipulate data. Most of the Invoice Transaction functionality – getFee(), getTotalCost(), and other functions which got numerical totals for Invoices – was folded into the Invoice Class. Taxes are now contained in the Equipment, License, and Consultation classes, while the Government compliance fee was moved to the Government class and then called through the Invoice class itself.

Each of our classes initially had two constructors – the standard generic constructor and one which built from a string array (the better to read in data from a file, as was necessary in the earlier phases of this project). This functionality was lost in phase 3 and each class now contains only a single generic constructor.

Our initial database design contained extra tables for each subclass: Licenses, Consultations, Equipments, Governments, and Corporates. These last two held no additional information – only a primary key and a foreign key representing a Customer. Since each Customer already contains a customerType, these tables were determined to be redundant. The extra information contained in the Product subclasses was folded first into Products, then into ProductsInInvoices, since many of the fields (numberOfUnits for Equipments, beginningDate for Licenses, etc.) were only relevant to specific instances of a Product in an Invoice.

# Bibliography

[1] ObjectAid LLC. The objectaid uml explorer for eclipse, March 2016.