|  |
| --- |
| Department of computer science & Engineering  University of Nebraska—Lincoln |
| Invoice System |
| Computer Science II Project |
|  |
| **Alex Linneman** |
| **4/18/19**  **Version 4.0** |

|  |
| --- |
| This document contains an in-depth overview of a software design written and developed in Java for the new invoice system constructed for Cinco Computer Consultants (CCC). |

# Revision History

[This table documents the various major changes to this document]

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Description of Change(s) | Author(s) | Date |
| 1.0 | Initial draft of this document (Phase I) | Alex Linneman | 2019/1/31 |
| 2.0 | Initial draft of this document (Phase II) | Alex Linneman | 2019/2/14 |
| 3.0 | Initial draft of this document (Phases III & IV) | Alex Linneman | 2019/3/30 |
| 4.0 | Final draft of this document (Phases V & VI) | Alex Linneman | 2019/4/18 |

Contents

[Revision History 1](#_Toc6471110)

[1. Introduction 4](#_Toc6471111)

[1.1 Purpose of this Document 4](#_Toc6471112)

[1.2 Scope of the Project 4](#_Toc6471113)

[1.3 Definitions, Acronyms, Abbreviations 4](#_Toc6471114)

[1.3.1 Definitions 4](#_Toc6471115)

[1.3.2 Abbreviations & Acronyms 5](#_Toc6471116)

[2. Overall Design Description 5](#_Toc6471117)

[2.1 Alternative Design Options 6](#_Toc6471118)

[3. Detailed Component Description 6](#_Toc6471119)

[3.1 Database Design 6](#_Toc6471120)

[3.1.1 Component Testing Strategy - Database 8](#_Toc6471121)

[3.2 Class/Entity Model 8](#_Toc6471122)

[3.2.1 Component Testing Strategy – Java Classes 10](#_Toc6471123)

[3.3 Database Interface 10](#_Toc6471124)

[3.3.1 Component Testing Strategy – Database Interface 10](#_Toc6471125)

[3.4 Design & Integration of Data Structures 10](#_Toc6471126)

[3.4.1 Component Testing Strategy 11](#_Toc6471127)

[3.5 Changes & Refactoring 11](#_Toc6471128)

[4. Bibliography 12](#_Toc6471129)

# Introduction

The application presented in this document is an invoice system created for Cinco Computer Consultants. This new invoice system is to replace the original, outdated invoice system. CCC offers many goods and services to its customers and its product line and clientele are only growing, which calls for a demand for an application to effectively and efficiently handle the data involved in an invoice.

The object-oriented Java application will be able to process data files from customers and products alike as well as incorporate the old data files in with the new when producing invoices. It will be tailored to CCC and implement their business model and rules.

## Purpose of this Document

The purpose of this document is to explain the integration of the invoice system into CCC’s business as well as outline the specifics and details of the application itself. This document should help one better understand the details of the application and the logistics of it.

## Scope of the Project

There is a great deal of information that goes into an invoice. Each product has a unique alphanumeric code and name, information on the customer, information on the salesperson, and the number of products in an order are all items of information that must be included in an invoice, to name a few. Backed by a database, the application will process and incorporate data into invoices. Because it is a new generation invoice system, the application must be made to be compatible with the original stored data files.

## Definitions, Acronyms, Abbreviations

### Definitions

Abstraction: representation of necessary components without the need for explanation or background information

Encapsulation: combination of segments of code that introduce a new entity and is characterized by grouping of data, protection of data, and grouping of methods that act on data

Inheritance: allowance of a class to extend commonalities for a more specific purpose

Polymorphism: ability of a single interface to accept many forms of objects and types

### Abbreviations & Acronyms

ADT – Abstract Data Type

CCC – Cinco Computer Consultants

CRUD – Create, Retrieve, Update, Delete

DRY – Don’t Repeat Yourself

EDI – Electronic Data Interchange

JDBC – Java Database Connectivity

JSON – JavaScript Object Notation

XML – Extensible Markup Language

# Overall Design Description

This application is created around the premise of object-oriented programming; therefore, it will utilize constructors to create objects and methods custom made for CCC. In order to transport objects from system to system the utilization of EDI will be performed by placing objects in file formats that are blind to platform identity, thus they are independent from one specific programming language. Two file formats of that nature are XML and JSON. So, the use of EDI will allow for the application to be virtually universally compatible.

The program itself will handle the parsing of data of all kinds that pertain to an invoice. It will go through the data files, find a common delimiter, separate each piece of information, and thoughtfully sort the formatted data.

The data will also be backed by a database. Using JDBC to query and extract the necessary data from the database, invoices can be created in a more modern way. The database will also allow for full CRUD functionality of the data, allowing not only the retrieval of it but also the insertion, updating, and destruction of it.

All of the data extracted via methods described above will then be used to create invoice reports complete with customer information, salesperson information, itemization, and price calculations. These reports will be well-formatted, and thoughtfully developed.

## Alternative Design Options

Ideas pertaining to application design that were entertained but ultimately rejected were:

* The initial draft of the program contained several classes for the use of constructors and unique objects. This was rejected due to the fact that it had no superclass and thus, no inheritance.
* The second initial draft of the program had far too much functionality in the main classes, thus breaking abstraction and encapsulation.
* Another draft of the program relied only on information from .dat files to create invoices but was changed to a reliance on a database due to the versatility it allows.

# Detailed Component Description

Many classes and database entries contribute to the overall functionality of the program. This section will detail each Java class and SQL database table along with its contents and functionality. This will be accomplished through a UML diagram for the Java classes and an EER diagram for the database tables alongside written explanation.

## Database Design

The database tables and their contents can be seen represented visually by Figure (1). The tables were based on the Java classes and can be viewed as a broader description of them. Each table has a unique integer primary key entry to allow for versatility in joins and other querying.

The parent table in this database is the Purchase table. The main functionality of the Purchase table is to store the purchases from each invoice. As a result, the Purchase table references the Invoice and Product tables directly mainly to establish a total cost and assign that cost to an invoice. The Invoice table mainly represents the customer and the salesperson, so it references the Customer and Person tables.

Due to the many references of primary keys through foreign keys, it was imperative that the database be constructed thoughtfully and meticulously. With that being said, it must also have a relatively simple way of being cleared of entries and could not be too complex so as to make the “delete” aspect of CRUD feasible.

A screenshot of a cell phone

Description automatically generated

Figure 1: Entity-Relation diagram representation of the database

### Component Testing Strategy - Database

Each table was immediately queried to show results upon the creation and insertion of data. Many of the same queries were used in JDBC from within the Java classes to ensure similar results. Once the testing was completed, full implementation was then permitted.

## Class/Entity Model

Due to the functionality demand of this application, specific organization must be followed. Each class and how they relate to each other can be found pictured in Figure (2). For every new, major piece of data or functionality addition, a Java class was made. Within the Java classes the DRY principle was used to create and implement methods in order to further establish the desired organization. For object assignment after reading in data, there are five main classes: Address, Person, Customer, Product, and Invoice.

The Address class exists for convenience when reading in data. It allows for the utilization of a constructor to fill the Address object with all the necessary information pertaining to a particular address. This object can then be easily placed in the constructor of a different class to associate the address record for a customer or person.

Beneath the Product class are three subclasses: ProductConsultation, ProductEquipment, and ProductLicense. Each subclass inherits the product code, type, and name from the Product class. Consultations are unique in that they have an additional consultant person code and an hourly fee. Equipment is unique in that they have an additional price per unit. Licenses are unique in that they have an additional service fee and an annual license fee.

In order to fill each of the objects with data input, the DataReader class is utilized. The DataConverter class will take lists of data and convert them into files with JSON format. The Database class acts as a JDBC connection factory from the SQL database. It has methods and information to connect to and disconnect from the database. The InvoiceReport class produces a neatly formatted, itemized invoice for every record of an invoice as well as the batch totals.

A close up of text on a white background

Description automatically generated

Figure 2: Unified Modeling Language representation of the Java classes

### Component Testing Strategy – Java Classes

As a precautionary measure, each calculation, means of obtaining information, or method implementation was first tested manually with a handwritten test case. If the program output matched the test case without any discrepancies, the program was deemed to be suitable and correct for the purpose it served. The program was also tested against corner cases and forced to fail in order to observe its behavior in these situations.

## Database Interface

The utilization of JDBC to interact with the database begins and ends in the Database class. Each constructor class has methods to query the database and store information from the query. Most of these constructor classes have two JDBC methods: one that accepts an integer for an identifier number of a specific entry and returns that specific entry and one that returns an entire list of all database table entries as objects from the constructor. These methods utilize the Database.openConnection() and Database.closeConnections() methods to access and cancel database connections before and after the database is queried for desired results.

### Component Testing Strategy – Database Interface

The main, InvoiceReport, was created before the implementation of JDBC, so as long as the input data for the program remained the same, the results should be the same between reading in files or database entries. So, the ultimate test of using JDBC was if it could produce the same results as before in the InvoiceReport class. After each method was created, it was tested to see if it would return the desired contents in a simple test case, until each method was finally implemented. When the methods replicated the original InvoiceReport results and endured corner cases, they were accepted.

## Design & Integration of Data Structures

In the final phase of the application design, a sorted list ADT is incorporated and utilized to hold an arbitrary number of Invoice objects. This list of Invoices is sorted using the Comparable interface, allowing for any specific sorting to take place. In the Invoice class, there are methods and comparators to sort retrieve a sorted list in three ways: 1) by the name of the customer on the invoice, 2) by the invoice total, and 3) first by the type of customer (government or corporate), and then by the last name and first name of the salesperson. Ultimately, the benefits of the sorted list ADT come when producing the invoice reports, giving the user the power to sort the reports on a specific criterion or criteria.

The list ADT is an array-based list with very similar functionality to Java’s ArrayList. The list class, CCCList, takes advantage of its binarySearch() method to sort elements upon insertion, rather than sorting the elements after the creation of the list. This way, order is always maintained. A binary search implementation proved to be an efficient candidate for this as it has a criterion of being ordered and order is maintained throughout the process. The CCCList.binarySearch() method takes the current list, the element to be added, and a comparator as input. It then uses the specified comparator to compare list elements with the element to be added and returns the index where the new element belongs. In order to add the new element to the list, the add() method that takes an index and an element as input is utilized.

### Component Testing Strategy

The implementation of this phase of the application did not require a complete overhaul of its functionality. Rather, only some method inputs and return types were altered to handle the new list ADT. Thus, given that the application went through rigorous testing in producing invoice reports in earlier phases, the main testing done was producing the same invoice reports as before but in specified orders. On top of that, several test cases that would knowingly produce errors were passed through the application to observe how the list ADT would handle exceptions. After troubleshooting and assigning messages to exceptions, testing methods upon implementation, and checking for discrepancies in calculations between this phase and earlier phases, the list ADT was accepted.

## Changes & Refactoring

In Phase I, constructors were defined and data was read in from .dat files converted to JSON format and stored in .json files inside of a main.

Phase II added inheritance between constructors, added an Invoice class, and created a class for reading in data from .dat files. It used a main to perform calculations and produce invoice reports.

Phase III created a database corresponding to each constructed object in the Java project.

Phase IV utilized abstraction and encapsulation to remove the calculations from the invoice report main, and rather perform them in the Invoice class as methods. Various other helper methods were included to reduce the functionality of the main and produce more readable code. JDBC was introduced in order to extract the data from a database and produce the invoice reports from database data.

Phase V implemented a new class, InvoiceData.java, for adding and removing database entries. In order to add and remove database entries more efficiently, changes were made to the database tables. For example, the Purchase table was created to keep product details separate, but joined with a foreign key, from the Invoice table which then stored information on the customer and salesperson.

Phase VI enjoyed all benefits of the tweaking of earlier phases and only implemented a list ADT with sorting algorithms and comparators to produce sorted invoice reports.

# Bibliography

[1] Bourke, Chris. *Computer Science II*. 2017.

[2] “Class Diagram.” *ObjectAid UML Explorer*, ObjectAid LLC, [www.objectaid.com/class-diagram](http://www.objectaid.com/class-diagram).

[3] “Class Collections.” *Collections (Java Platform SE 7 )*, Oracle, 6 Oct. 2018, [docs.oracle.com/javase/7/docs/api/java/util/Collections.html](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html).