|  |
| --- |
| Department of computer science & Engineering  University of Nebraska—Lincoln |
| CEG Invoice Management System |
| CSCE 156 – Computer Science II Project |
|  |
| **Grant Harrison, Sean Mitchell** |
| **11/11/2016**  **[Document Version 4.0]** |

|  |
| --- |
| This document details the new Invoice Management System for the Cineclark entertainment group, implementing an extensible Java-based approach utilizing a MySQL database. |

# 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 design document | Grant Harrison, Sean Mitchell | 2016/9/16 |
| 2.0 | Phase 2 | Grant Harrison, Sean Mitchell | 2016/10/7 |
| 3.0 | Phase 3 | Grant Harrison, Sean Mitchell | 2016/10/21 |
| 4.0 | Phase 4 | Grant Harrison, Sean Mitchell | 2016/11/11 |

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 4](#_Toc349390343)

[2.1 Alternative Design Options 4](#_Toc349390344)

[3. Detailed Component Description 4](#_Toc349390345)

[3.1 Class/Entity Model 4](#_Toc349390346)

[3.1.1 Component Testing Strategy 5](#_Toc349390347)

[3.2 Class/Entity Model 5](#_Toc349390348)

[3.2.1 Component Testing Strategy 5](#_Toc349390349)

[3.3 Database Interface 5](#_Toc349390350)

[3.3.1 Component Testing Strategy 5](#_Toc349390351)

[3.4 Design & Integration of Data Structures 5](#_Toc349390352)

[3.4.1 Component Testing Strategy 5](#_Toc349390353)

[3.5 Changes & Refactoring 5](#_Toc349390354)

[4. Additional Material 5](#_Toc349390355)

[5. Bibliography 6](#_Toc349390356)

# Introduction

This project is a new invoice management system for the Cineclark entertainment group (CEG), replacing the old flat-file based invoice management system. The new CEG Invoice Management System is a Java based, object-oriented design. It is capable of uploading the old data from flat files, modeling it in Java classes, generating appropriate XML documentation, and finally storing and retrieving the data from a MySQL database.

The Cineclark entertainment group has two broad categories of products – tickets and services – and has two broad categories of customers – students and general customers. Of these products, season-passes and movie-tickets are considered tickets, and parking-passes and refreshments are considered services. Each of these broad categories, tickets and services, is taxed differently and have different business rules applied.

When a sale is made on the new CEG Invoice Management System, the person who made the sale, the customer purchasing the products, and the products themselves are all recorded in the invoice. The invoice also contains relevant information like an invoice code, a salesperson code, and the date the sale occurred. Then, the invoice is stored in a database along with lists of the customers, salespersons, and products involved in all transactions with the company.

## Purpose of this Document

This document is intended to show the design, testing, and implementation of the new CEG Invoice Management System. It gives all components and elements of the design and explains each phase in detail.

## Scope of the Project

This project will design and implement a simple invoice system to replace an old system. Cineclark entertainment group is a popular movie theater chain that needs to update their services managed by the old invoice system. Their old invoice system is based on flat files and is no longer feasible to continue to use. Cineclark wants an object oriented system (OOS) design from scratch that is written in java. The system has to support the company’s business model by implementing their business rules and providing the functionality which is the generation of reports managing customers.

## Definitions, Acronyms, Abbreviations

### Definitions

XStream – a collection of libraries used to generate platform-independent XML and JSON

Encapsulation – data hiding, preventing outside applications from directly accessing data in an object or class.

Class – a programming concept that provides a template for creating objects, containing fields (which contain data) and methods (which provide instructions for handling data).

Constructor – a special method within a class that creates an object from that class.

Inheritance – deriving a subclass from a superclass, where the subclass expands upon the superclass, potentially adding new fields and methods while retaining all the functionality of its superclass.

Polymorphism – the ability for object or method to express many types.

Abstraction – implementation hiding, this prevents users from seeing various entities in an API are actually coded, and shows only essential information to a user.

Normalization – a term which refers to the process by which databases are made efficient in their organization. There are three “Normal Forms” which guide this process.

First Normal Form (1NF) –a table should eliminate duplicative columns.

Second Normal Form (2NF) – a table should replace duplicative rows in a table with the same value with a reference to another table which contains that value only once.

Third Normal Form (3NF) – a table should not store anything that doesn’t depend upon the primary key.

### Abbreviations & Acronyms

EDI – Electronic Data Interchange

XML – Extensible Markup Language

OOP – Object Oriented Programming

DDL – data definition language

FK – Foreign Key

JSON – JavaScript Object Notion

FK – Foreign Key

SQL – Structured Query Language

UML – Unified Modeling Language

API – Application programming interface

JDBC – Java Database Connectivity

ADT – Abstract Data Type

# Overall Design Description

Broadly speaking, the API is charged with doing five main tasks: reading information, modeling this information with classes, creating objects using the classes and data, implementing business logic, updating the database, and producing output. Each of these tasks and the relevant portions of the API and database are detailed below.

The API must first read information from various sources; either the relational database or from flat files. SQL commands are broadly separated into two categories: updates and queries. The Database class provides basic methods to establish a connection to the database and perform select SQL queries relevant to object creation. In addition, a FlatFileReader and InvoiceReader class are provided for reading from flat files.

Next, the program models real-world entities through java classes. At the top is an abstract Record class, which helps implement business logic. After this, a Product, Customer and Person class broadly model their real world counter parts. These each, in turn, have the following subclasses: Product has Ticket and Service subclasses, with further subclasses MovieTicket, SeasonPass, Refreshment, ParkingPass. Customer divides itself into General and Student subclasses. And Person has no subclasses. Further, there are two classes which function solely in an aggregation relationship – Email and Address – and are used by the Person and Customer classes. Each class properly maintains data encapsulation and abstraction. Furthermore, the class hierarchy is logically designed to allow for useful polymorphism and inheritance.

With data and classes, the project creates objects with several factory classes for the objects above. Business logic can then be implemented by DataConverter and Invoice classes. Should data need to be changed, an InvoiceData class provides methods for generating SQL updates. Finally, output can be generated in the following ways: The InvoiceReport class generates invoices to the standard output; and the XMLWriter class generates an XML file modeling the general classes Product, Person and Customer and their respective subclasses (but not the specific objects).

## ­Alternative Design Options

## Another design option would be to implement a procedural programming approach. Although this would be difficult, the program would start with a procedure, which is a sequence of statements. For example, imperative statements such as, assignments, tests, loops and invocations of sub procedures are all examples of procedural programming. However, using OOP (Object Oriented Programming) is more useful because it uses abstraction in the form of classes and objects to create models based on real world environment.

Another design option for the SQL database structure would be to normalize the Address table further by creating a County table. This would prevent redundancies in the Address table by allowing simply a countryID to be listed, rather than a string with the country name. Not only does use less memory, but it also normalizes the database in accordance to the Second Normal Form (2NF).

## Detailed Component Description

In the first phase design and implement objects that will form a basis for the system and create parsers to read data from flat files. In the second phase define the objects and define relationships between them to generate a summary and a detailed report that aggregates pieces of data together. For the third phase the focus on designing a relational database to model objects and manage data, and for the fourth phase re-factor code to load and persist objects to database.

Going more in depth about the first phase, create parsers to read the data file, and to create those parsers use json or xml. At design time get the description of the file structure and generate code for the objects that represents the data and parser. The parse will happen at run time and will populate the java classes.

For the project it is recommended to use OOP practices, and demonstrate the principles abstraction, encapsulation, and polymorphism. Using OOP practices it is also recommended to use bottom- up design and identify the entities and design classes that can be used as the building blocks to implement the larger application.

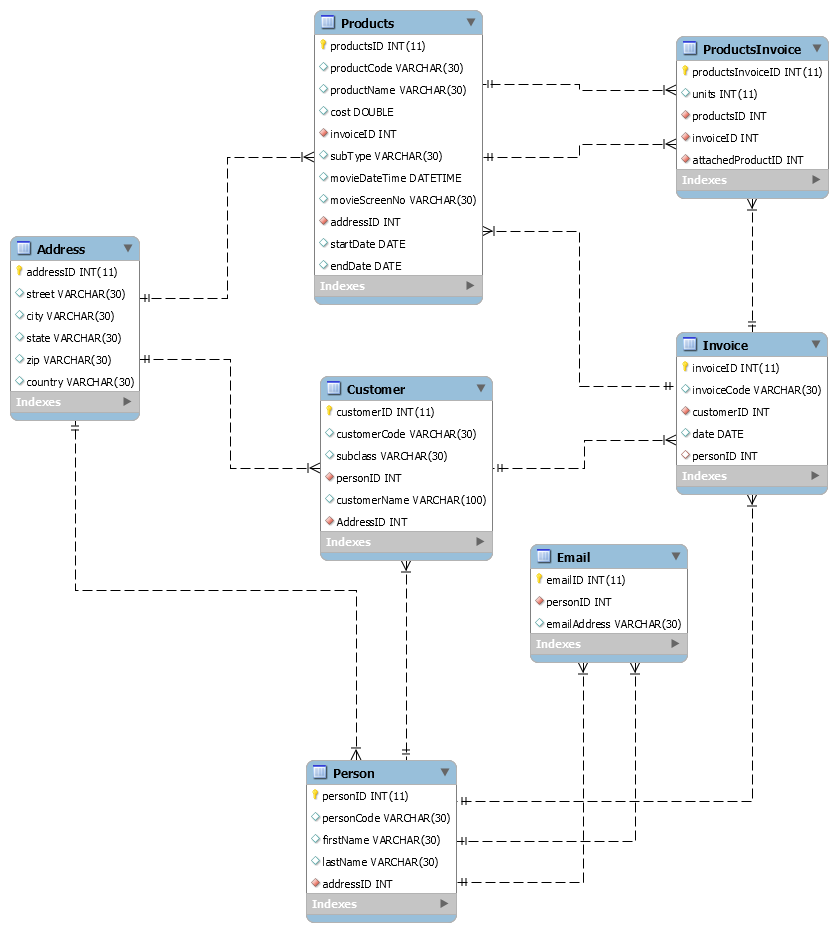
For phase 2 the project need functionality added to the classes that was created in Phase 1, and also needs to design new classes to complete the core functionality of the invoice system. To produce the two detailed and summary reports the projects needs to integrate all of the classes. The first report is a summary report that will report overall figures and totals. The second will report the details of each invoice. Goal is to continue to utilize polymorphic behavior to simplify the code it is also recommended to stay away from handling similar objects in a dissimilar manner since they should have properly defined a common public interface.

## Database Design

For the database design a data model to support the previous application, meaning implement an SQL database. The database needs to have tables designed to support data related to the entities in the previous phase as well as the relationships between these entities.

The database is created using MySQL and the database needs to be implemented using a DDL file (which is just a plain text file containing SQL queries). It should have the ability to create, retrieve, update, and destroy data as needed, while maintaining data integrity. This data integrity is maintained by forcing uniqueness constraints on fields that conceptually should not have duplicates. The database structure will need to have tables created for each of the major data types such as Products, Invoice, Customer, Person, Email, and Address. Most tables will have a relationship between them through joined tables and foreign keys.

For the database design it is recommended to use many tables like Address, Customer, Email, Invoice, Persons, and Products to accomplish successfully writing their queries. To name a few columns the database could have, for example would be street, city, and zip for Address table and Persons table included columns like person code, first name, last name, and address id. In every table there needs to be a primary key created and that is just something unique about every table. The naming convention for the primary key is usually the table name and the “id” after it. Also there will be many tables that contains foreign keys to make relationships between the tables. Some relationships may be many to one or many to many relationship. For example some relationships the database could have is, Products and invoice, Invoice to customer, and Customer to Person and Address, which would be a one to many relationship. Because there so many ways to design the database some columns may be considered “unnecessary or necessary”, however the database should strive for a design that makes sense and be easier to write queries for.



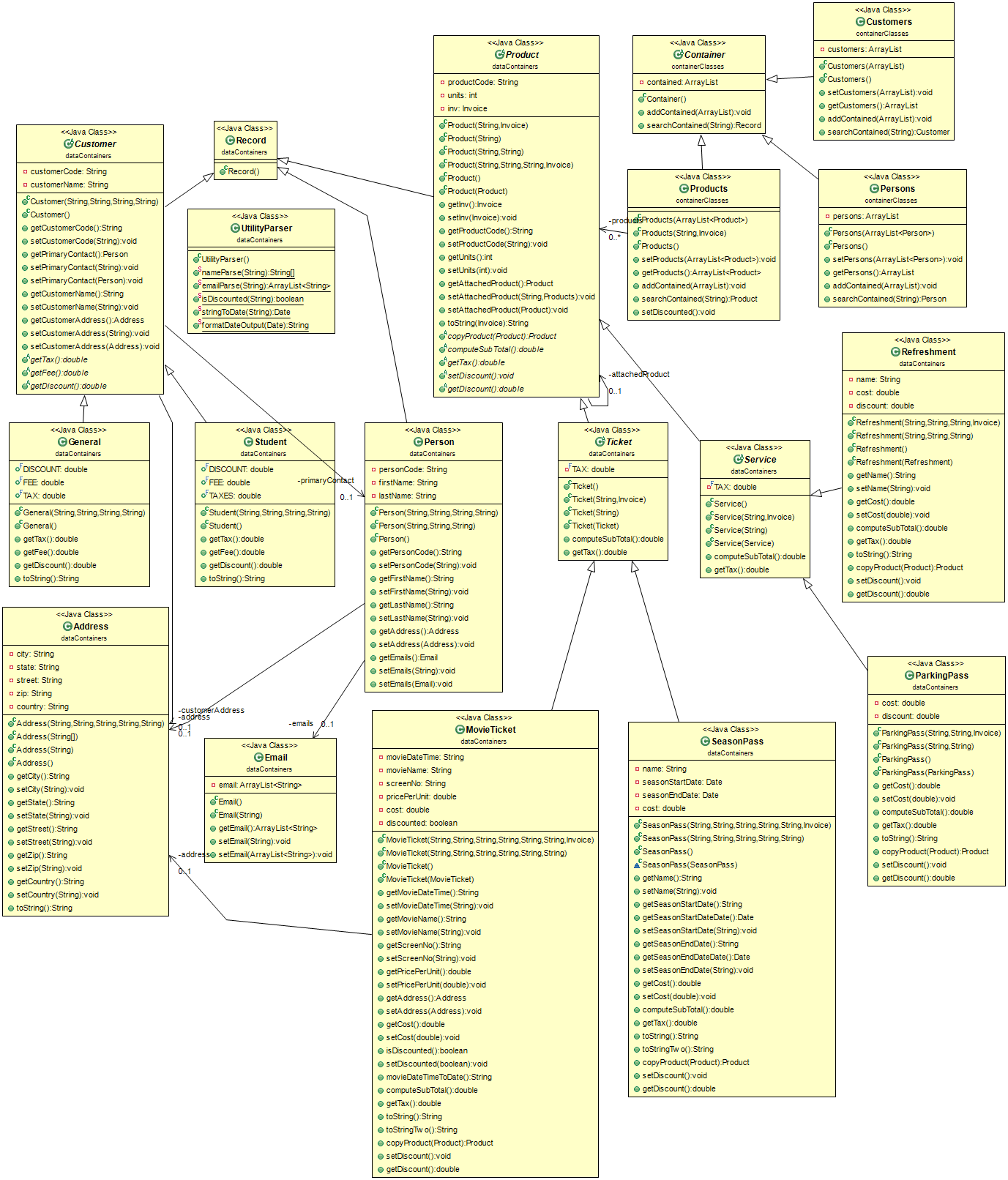
### Component Testing Strategy

Created many data files such as product, customer, and person file to test program locally once each file reader was able to read each data file correctly. Designed non trivial cases to demonstrate the program was tested locally to some degree and also needed independent input-output. Test cases were meant to probe and break “bad” code, but stay within the requirements specified.

## Class/Entity Model

For phase 4 the design will be modifying the applications in phase 1 and 2 to interact with the database in phase 3. The application will be modified to persist data into the database and load data from it rather than from the local flat data files. More specifically implement an API to interact with the database using JDBC. The API will provide methods to load and persist data on the database.

The reports need to be sorted by total (highest to lowest) so the design will have needed to have created a sorted list by ADT or a linked list for the ordering. The ADT or linked list is to facilitate adding, removing, and retrieving/iterating over elements. The order should be maintained, not imposed by a method call. So the design should use a constructor to maintain the ordering in the list implementation and should use generics by parameterizing it as well. The design should not use any standard JDK collections objects or algorithms nor exploit functionality provided by MySQL database or implement a sorting algorithm outside the class list. So for using OOP practices it is necessary to define and implement a list that sorts by the same ordering throughout its life cycle.

As far as implementation goes the main drive class (Invoice Report Class) will still keep its functionality, however instead of reading from data file it will be making a connection to the database, and load the appropriate data and create the objects. It is recommended to implement and reuse several factory methods that retrieve instances of the defined classes by loading from the database

Lastly we need to design converter class. In this class a FlatFileReader object reads data from the flat file. This also creates objects and stores the objects in an object array list and returns the array list of objects. Then, an XML object is required. This will write the array list into an XML file.



For phase 2 were adding functionalities to the classes such as methods for customer and product. For the customer we need to make this class abstract and declare some abstract methods such as get tax, get discount, and get additional fee. We also need to define two types of customers which is general and student. The student type of customer is exempt from taxes, resulting in an 8% discount and a $6.75 processing fee. For the product class we need to declare it abstract as well and defined some abstract methods such as compute subtotal, get tax, and compute grand total. For the new class we are creating, which is an invoice class and an invoice FlatFileReader, to read the invoice data file. The invoice data file is a little more complicated to read. For the invoice data file there is an invoice code, customer code, salesperson code, and invoice date all separated by a semi colon. Invoice data file also has a comma delimited product list. Each product code is either a Movie-Ticket, Parking-Pass, or Refreshment. After the product code there is a single number representing the number of units. With this information about the invoice data file in mind we have an idea of some attributes and methods. Some methods for the invoice class involve methods for computing total cost, taxes and fees, and a method for generating formatted report output.

### Component Testing Strategy

For this component testing strategy the project should use MySQL to thoroughly test the database design. More specifically the test cases contained several queries to test the design and its functionality. The goal is to create a design flexible enough that records can be easily added/modified/removed without data integrity problems.

## Database Interface

For phase 4 the designer will be modifying the applications they designed in phase 1 and 2 to interact with the database in phase 3. The application will be modified to persist data into the database and load data from it rather than from the local flat data files. More specifically the designer will implement an API to interact with their database using JDBC. The API will provide methods to load and persist data on the database.

The reports need to be sorted by total (highest to lowest) so the designer will have will have to create an sorted list by ADT or a linked list for the ordering. The ADT or linked list to facilitate adding, removing, and retrieving/iterating over elements. The order should be maintained, not imposed by a method call. So the designer should use a constructor to maintain the ordering in the list implementation and should use generics by parameterizing it as well. The designer is not allowed to use any standard JDK collections objects or algorithms nor exploit functionality provided by MySQL database or implement a sorting algorithm outside the class list. So for using OOP practices it is necessary to define and implement a list that sorts by the same ordering throughout its life cycle.

As far as implementation goes the main drive class (Invoice Report Class) will still keep its functionality, however instead of reading from data file it will be making a connection to the database, and load the appropriate data and create the objects. The designer is recommended to implement and reuse several factory methods that retrieve instances of the defined classes by loading from the database

### Component Testing Strategy

For this component testing strategy we used MySQL to thoroughly test our database design. Our test cases contained several queries to test our design and its functionality. Our goal was to create a design flexible enough that records can be easily added/modified/removed without data integrity problems.

## Design & Integration of Data Structures

For the API can either use a LinkedList or ADT. The ADT can hold any numbers of any types of objects through parameterized polymorphism. Also the ADT is linked based so it only needs to keep track of the head and because the list is based on references from nodes to nodes the list is automatically resized. However, the ADT list does have some limitations. The ADT resizing can be expensive and has many other limitations like new memory allocation. To add or delete an element to 0th position, one must move every element in the array up or down one position. For LinkedList there are no fixed size or resizing, operations involve only shuffling references around. However, there is no random access functionality, unlike the ArrayList. For LinkedList need to know where the end (or tail) of the list is.

### Component Testing Strategy

For this component testing strategy create different instances of ObjectList. The LinkedList or ADT replaces the ArrayList used in the previous development. The testing strategies are similar to those outlined in **Section 3.2.1,** however outputted invoice has a certain order (highest-to-lowest). So if the reports are the correct order, it can be assumed that the ADT or LinkedList is functioning properly.

## Changes & Refactoring

Before phase 1 was finished, the FlatFileReader class was altered to accept multiple emails for a given Person object. Other bad code was changed because of non-trivial test cases which did break bad code. After phase 1, comments were added to the majority of the code and redundant code was removed.

After phase 3, the database was altered to include a “ProductsInvoice” mapping table which joined together the Products and Invoice tables. This allows for much easier maintenance of the various Products classes and their objects. Furthermore, the movieAddress column of the Products table was changed to simply be a foreign key referencing the Address table, where the address was added in. This eliminated duplicate data in the rows of the table, obeying the Second Normal Form (2NF).

# Bibliography

[1] W3schools. SQL Functions. Retrieved from October 21, 2016, from

<http://www.w3schools.com/Sql/sql_functions.asp>

[2] Oracle java Documentation. The Java Tutorials (Inheritance). Retrieved from September 14, 2016, from

<http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>

[3] Oracle java Documentation. The Java Tutorials (JDBC Database Access). Retrieved from November 10, 2016, from

<http://docs.oracle.com/javase/tutorial/jdbc/>