|  |
| --- |
| 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 shows the sequence of alterations made to the project.

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Description of Change(s) | Author(s) | Date |
| 1.0 | Phase 1 | 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](#_Toc466587870)

[1. Introduction 3](#_Toc466587871)

[1.1 Purpose of this Document 3](#_Toc466587872)

[1.2 Scope of the Project 3](#_Toc466587873)

[1.3 Definitions, Acronyms, Abbreviations 3](#_Toc466587874)

[1.3.1 Definitions 3](#_Toc466587875)

[1.3.2 Abbreviations & Acronyms 4](#_Toc466587876)

[2. Overall Design Description 4](#_Toc466587877)

[2.1 Alternative Design Options 5](#_Toc466587878)

[Detailed Component Description 5](#_Toc466587879)

[3.1 Database Design 6](#_Toc466587880)

[3.1.1 Component Testing Strategy 7](#_Toc466587881)

[3.2 Class/Entity Model 8](#_Toc466587882)

[3.2.1 Component Testing Strategy 11](#_Toc466587883)

[3.3 Database Interface 11](#_Toc466587884)

[3.3.1 Component Testing Strategy 12](#_Toc466587885)

[3.4 Design & Integration of Data Structures 12](#_Toc466587886)

[3.4.1 Component Testing Strategy 12](#_Toc466587887)

[3.5 Changes & Refactoring 12](#_Toc466587888)

[4. Bibliography 12](#_Toc466587889)

# 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 divides customers into two categories – 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 movie theater chain in need of an update to 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 programming (OOP) 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 generates reports managing invoices.

## 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 how 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 Notation

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, and ParkingPass. Customer divides itself into General and Student subclasses. 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 application 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 the 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 of the country name. Not only does this use less memory, but it also normalizes the database in accordance to the Second Normal Form (2NF).

## Detailed Component Description

Phase 1 of the project requires the design and implementation of objects that will form a basis for the system and create parsers to read data from flat files. In the second phase, the project defines the objects and defines relationships between them to generate a summary and a detailed report that aggregates pieces of data together. For the third phase, a relational database is made to model objects and manage data. Finally, for the fourth phase, an Abstract Data Type (ADT) LinkedList is created, along with factory methods for creating objects from the database. This, in turn, allows a fully sorted list of invoices to be presented.

Going more in depth about the first phase, the design created parsers to read the data file, and also created parsers for XML. At design time the project retrieved the description of the file structure and generated code for the objects that represented the data and parser. The parse happened at run time and populated the Java classes.

The project utilized OOP practices, and demonstrated the principles of abstraction, encapsulation, and polymorphism. Using the OOP practices the program used bottom-up design and identified all the entities and design classes that could be used as building blocks to implement the larger application.

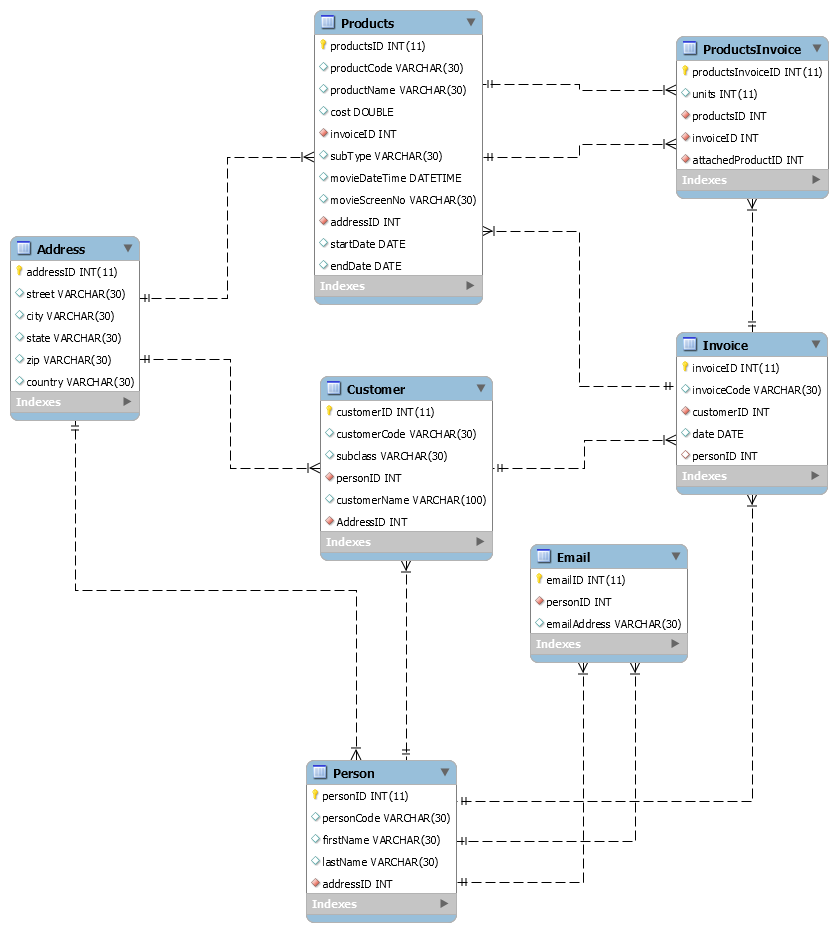
For phase 2, functionality was added to the classes in Phase 1, and new classes were designed to complete the core functionality of the invoice system. To produce the first report (a summary report that will report overall figures and totals) and the second report (the details of each invoice) the project integrated all of the classes that were made in the previous phases. One of the project goals was to continue to utilize polymorphic behavior to simplify the code and stay away from handling similar objects in a dissimilar manner.

## Database Design

For the database, a schema was designed to support the previous application. This meant an SQL database was implemented. 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 was created using MySQL and implemented using a DDL file (which is just a plain text file containing SQL queries). It has 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 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. However, an additional table, called ProductsInvoice, was created as a mapping table to join Products and Invoice. Furthermore, all tables were designed to satisfy the principles of database normalization; that is to say, they avoid duplicate columns and rows, and only data relevant to the primary key is stored.

The database was designed 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 has, for example, are street, city, and zip for the Address table. The Persons table includes columns like personCode, firstName, lastName, and addressID. In every table, a primary key was created for unique identification of table rows. The naming convention for the primary key is usually the table name and the “ID” after it. Also, many tables contain foreign keys which link tables. Some relationships may be many-to-one or many-to-many relationship.

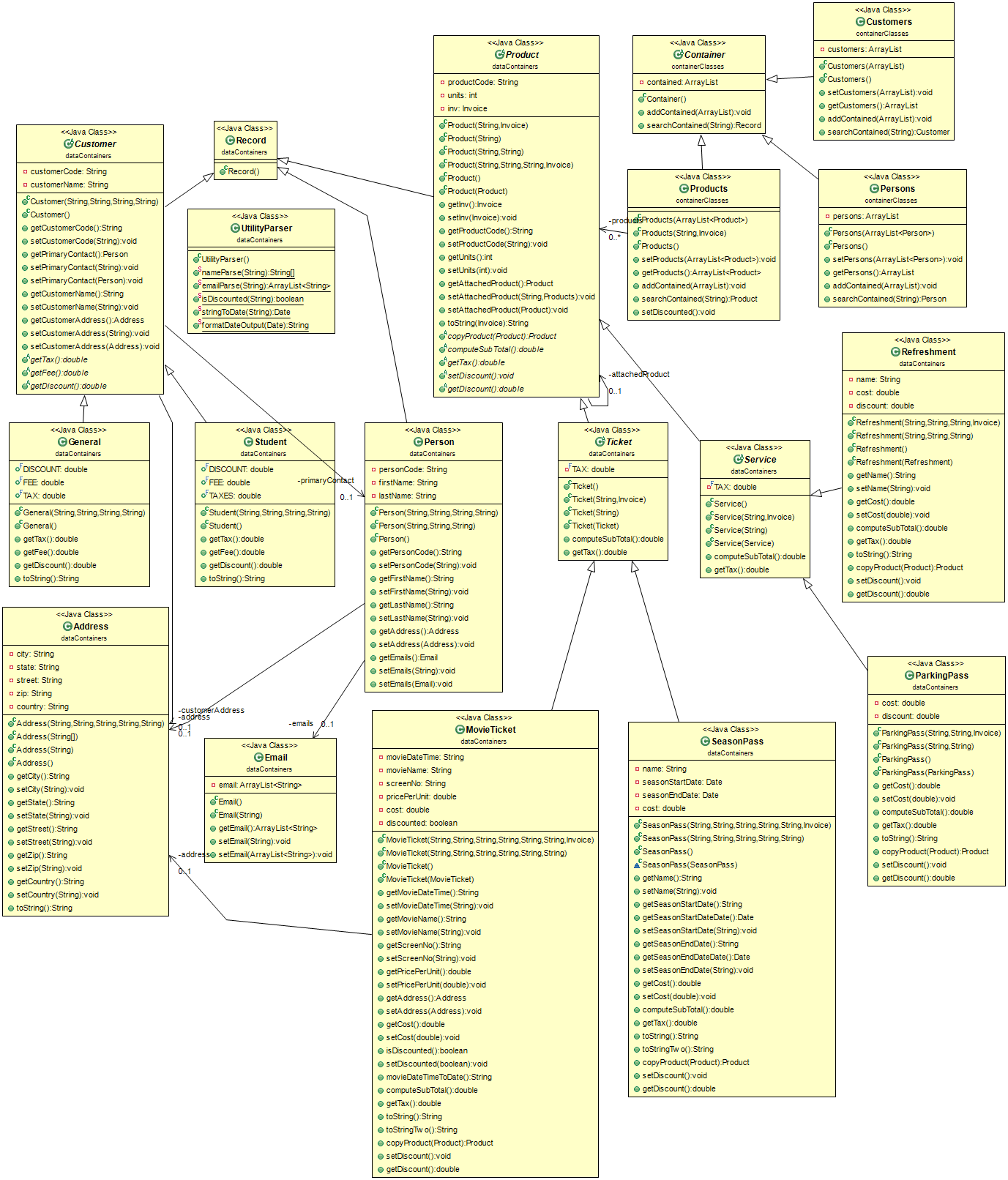


### Component Testing Strategy

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.

## Class/Entity Model

At the beginning of the project, simple data containers were made to hold the contents of three flat files: customers.dat, persons.dat, and products.dat. These were mapped to their appropriate classes and subclasses, as seen in the chart below. Later on, after phase 2, further functionality was added to the classes. In particular, custom *toString()* methods were added, as well as methods for getting some business-logic related values. Actual business logic was handled in the Invoice and InvoiceReport classes. This was a logical distinction, and fits well with established principles in OOP like agile.



To parse the data from flat files, a FlatFileReader and InvoiceReader class were created. These classes utilized a simple algorithm designed specifically for the formatting of the flat files to accurately parse the data. The DataConverter class was then responsible for taking this data and using it to create objects. Because storing these objects and manipulating them presented itself as a challenge using a standard ArrayList, special Products, Persons and Customers container classes were created that allowed these lists of objects to have certain methods called on them. This required a high level of abstraction in the code, and to accomplish this goal, a Record class, the superclass for all other classes, was created. With these objects created, the XMLWriter class uses the JAXB API to marshal the data into a new XML file.



For phase 2, further functionalities were added to the classes. The Product class was declared as abstract, and methods such as *getTax()*, *setDiscount()*, *computeSubTotal()* were added. Other classes, like Customer, Ticket and Service were made abstract. More details specific to business logic was added. Specifically, the Student type of Customer is exempt from taxes, resulting in an 8% discount and a $6.75 processing fee. Moreover, for the invoice data file fields such as invoiceCode, customerCode, salespersonCode, and invoiceDate were parsed from the flat files and added. Inside each Invoice is a Products object containing an ArrayList of Product objects. These Product objects were created from data given by the flat files. Each product code is either a MovieTicket, ParkingPass, SeasonPass, or Refreshment. After the product code is a single number representing the number of units. With this information about the invoice data file, objects can be instantiated. This was how the application generally handled the creation of objects and implementation of classes.

### Component Testing Strategy

To test this, test data files to create Product, Customer, and Person objects were uploaded. The expected results were hand calculated; and if the file reader was able to read each data file correctly, the code was deemed a success. During this stage of development, much time was spent improving the complex abstraction that made the DataConverter class function properly.

## Database Interface

For phase 4, the Java-based, well-encapsulated and abstracted program from phases 1 and 2 were then combined with the database designed in phase 3. The application was modified to load and retrieve data using the database rather than from the local flat data files. Furthermore, the design implements an API to interact with the database using JDBC. The API provides methods through the Database and InvoiceData classes to load data and alter it in the database.

Furthermore, the application utilizes a LinkedList built upon an Abstract Data Type (ADT) to sort reports by total sale volume (highest to lowest). This ADT LinkedList also facilitates the adding, removing, and retrieving/iterating over and of elements. The order was maintained, not imposed by a method call, meaning that order was kept as objects were added. A comparator class was created to maintain the ordering in the list implementation. The ADT LinkedList also uses generics by parameterizing it, making it highly extensible should features need to be added later.

The driver class has not changed; however, instead of reading from data file, the application will connect to the database and load the appropriate data to create the objects. To accomplish this goal, each main class that was modeled in Java-based OOP has a matching factory method, which provides a means for creating an object from the database. These methods rely on various methods from the Database class to retrieve data from the schema on MySQL to create the object.

### Component Testing Strategy

For this component testing strategy, various MySQL queries and updates were used to thoroughly test database design. A set of test data was uploaded, which contained data for invoices and products. These were then manipulated using a batch of queries and updates; and if the outputted results matched a pre-calculated value, the test was considered a success. This showed the design to be flexible enough that records can be easily added/modified/removed without data integrity problems.

## Design & Integration of Data Structures

To better manage the list of Invoice objects, the application implements a LinkedList of an Abstract Data Type (ADT). The ADT provides methods for adding and removing Invoices (or any object) to the list, as well as unique methods for testing for equality and placing the objects in a HashMap. The ADT is a linked list, meaning it is a collection of nodes, with one pointing to the next. This means the size of the list never has to manually be adjusted beyond simply adding another pointer to a new node or removing a pointer. Furthermore, this design allows for ease of insertion and deletion – all that is required is changing the references of a few nodes. This has notable advantages over a standard ArrayList based ADT. In this case, resizing the ADT List can be expensive and must be done periodically as the size of the array grows or shrinks. Further, to add or delete an element to 0th position, one must move every element in the array up or down one position. For a LinkedList there is no fixed size, operations involve only shuffling references around. However, there is no random access functionality, unlike the ArrayList. Still, because of the ease of use of a LinkedList and its flexibility, it was chosen to be the mode of organizing the collection of Invoices to be handled by the database.

### Component Testing Strategy

For this component, the testing strategy was to create different instances of the ADT LinkedList with different invoices inside. The ADT LinkedList replaces the ArrayList used in the previous development, and now contains the collection of Invoice objects. The testing strategies are similar to those outlined in **Section 3.2.1;** however, in this case, the test will be to show that the outputted invoice has a certain order (highest-to-lowest in total sale value). If the reports are the correct order, it can be assumed that the ADT 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/>