|  |
| --- |
| 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]** |

|  |
| --- |
| [Project to design and implement a simple invoice system to replace the existing system that company currently uses.] |

# 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 – concealing data fields within a class to prevent interference in these data fields by the outside world

Class – programming entity that models a real-world object, containing state and behavior.

Constructor – creates an instance of a class when it is called in the application.

Inheritance – the ability to extend the definition of objects; creating objects from previously created or defined objects.

Polymorphism – is the ability to create a variable, method, or an object that has more than one form (type).

Abstraction – the mechanism by which objects are given their representation and implementation details are hidden.

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

There are four phases that the project is broken down to. The first phase is Data representation & Electronic Data Interchange. The second is summary and detailed reports. The third phase is the database design and the fourth and final case is database connectivity and sorted list.

## Alternative Design Options

Another design option is would be using procedural programming approach. Although this would be difficult, you 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.

# Detailed Component Description

In the first phase the designer is expected to 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 the designer will 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 designer will focus on designing a relational database to model objects and manage data, and for the fourth phase the designer will be re-factoring code to load and persist your objects to your database.

Going more in depth about the first phase, the designer is expected to create parsers to read the data file, and to create those parsers the designer has a choice of using json or xml. At design time the designer will need to 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 the designer is expected to use OOP practices, and demonstrate the principles abstraction, encapsulation, and polymorphism. If the designer is using OOP practices its 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 designer will be adding functionality to the classes they have in Phase 1 and design new classes to complete the core functionality of the invoice system. Then the designer will integrate all of their classes to produce two reports. 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 also the designer should 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 the designer will design a data model to support the previous application, meaning they implemented an SQL database. The designer need to designed tables to support data related to the entities in the previous phase as well as the relationships between this entities.

The database is created using MySQL and the designer is expected to implement database using 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 conceptually should not have duplicates. The designer will need to create table 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 the designer should use many tables like Address, Customer, Email, Invoice, Persons, and Products to accomplish successfully writing their queries. To name a few columns the designer could have for the database, 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 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 designer 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 will be considered “unnecessary or necessary”, however the designer should strive for a design that makes sense and be easier to write queries for. Another design option for the database structure would have been to make mapping tables. For example an “Invoice and Products” mapping table would further data integrity and normalization.

### Component Testing Strategy

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

## Class/Entity Model

More specifically were designing classes that are simple data containers. The three data containers are customers.dat, persons.dat, and products.dat. For the persons data file we will create a person class and an address class. Person class has an address class as an attribute so were going to use the address class within the person class. For the customers data file were going to use persons and address class within customers class because it has persons and address class as an attribute. For designing the product class we have looked at movieTicket, SeasonPass, Parkingpass, and Refreshment to see each type have in common. The product has constructors and getters & setters methods involved. For example movie tickets has address attribute so the address class can be used there and season-passes has person class attribute. Parking pass has code and a parking fee. Does not look parking pass contains any of the classes previously written. Refreshments has the format of code name and cost. Persons class is an attribute of refreshments. Some classes we used inheritance which is a hierarchical organization of classes to help and avoid duplication and reduce redundancy. Sub classes inherits all the variables and methods of its immediate parent and its ancestors. For example product has movieTicket, SeansonPass, ParkingPass, and Refreshment as a subclass. Creating these sub classes are as simple as using the word “extends”. For example ParkingPass extends Product

Lastly we need to design converter class. In this class we need a FlatFileReader object to read 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. We also need to do design a json writer object or xml writer object. This will write the array list into a json or xml file. So the json writer class converts objects that read flat files and writes objects as a string into a 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

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.

## 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 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 Linklist or ADT replaces the Arraylist used in the previous development. The testing strategies is 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

Changed person class and file reader to be able to read 2 emails from a data file. Change code because of non-trivial test cases which were meant to probe and break “bad” code, but stay within the requirements specified. Later on changed database design to have an invoice products table. This made our database structure simpler and allowed us to make queries much simpler. So if the (ADT OR LinkedList) puts the invoices in the correct order and generates the summary and detailed reports from the database, in can be assumed that the “List” is working properly.

# Additional Material

[This is an optional section in which you may place other materials that do not necessarily fit within the organization of the other sections.]

# Bibliography

[This section will provide a bibliography of any materials, texts, or other resources that were cited or referenced by the project and/or this document. You *must* consistently use a standard citation style such as APA or MLA (good reference: <http://www.cws.illinois.edu/workshop/writers/citation/)>.]

[1] *Citation Styles*. (n.d.). Retrieved December 19, 2012, from [http://www.cws.illinois.edu/workshop/writers/citation/](http://www.cws.illinois.edu/workshop/writers/citation/))

[2] Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.