DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

Order-Invoice System

CSCE 156 – Computer Science II Project

Student 001

4/4/13
Version 5.1

This document describes the software design for the order-invoice system designed for Jenn-Eric Import/Export Company.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft	Student 001	2013/01/31
2.0	UML Diagram added, Acronyms added, More detailed description in several sections, Bibliography updated	Student 001	2013/02/14
3.0	UML Diagram updated, ER Diagram added, Bibliography updated	Student 001	2013/02/28
4.0	More detailed description in several sections, Bibliography updated	Student 001	2013/03/14
5.0	UML Diagram updated, ER Diagram updated, More detailed description in several sections	Student 001	2013/04/03
5.1	Final draft	Student 001	2013/04/04

Contents

1. Introduction

This is the Software Design Description for the Order-Invoice System designed for the Jenn-Eric Import/Export Company. It outlines the technical design of the application, which is being implemented to replace Jenn-Eric's current system, as well as outlining the database used to store Jenn-Eric's records related to order invoices.

1.1 Purpose of this Document

This design document serves as the outline for the Order-Invoice System. It gives all details with respect to the Java application, including detailing all classes used in the Order-Invoice application and their respective states and behaviors. This document also outlines the MySQL database schema used in the Java application, including details about the interaction of the application with the database.

1.2 Scope of the Project

The Order-Invoice System serves as an order manager for the Jenn-Eric Import/Export Company. The application produces summary reports of the company's orders. The application also produces detailed invoices of the orders for every customer. The application supports the company's current business model in the form of managing and applying the appropriate discounts, fees, and taxes to each customer and order. The Order-Invoice System does not take orders directly; instead, customers place orders through a third party system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Personal customer - An individual placing an order at the Jenn-Eric Company.

Academic customer - An educational institution placing an order at the Jenn-Eric Company.

Corporate customer - A private company or corporation placing an order at the Jenn-Eric Company.

1.3.2 Abbreviations & Acronyms

ADT - Abstract Data Type

API - Application Programming Interface

CRUD - Create, retrieve, update, destroy

ER diagram - Entity-relation diagram

JDBC - Java Database Connectivity

OOP - Object Oriented Programming

SQL - Structured Query Language

UML diagram - Unified Modeling Language diagram

XML - Extensible Markup Language

2. Overall Design Description

In regards to the application framework, the program is designed using the OOP design principles of abstraction, encapsulation, inheritance and polymorphism. This order-invoice system certainly lends itself to using objects. As such, the program features objects such as Orders, Products, Product Categories, Customers, Addresses, etc. Jenn-Eric has three different types of customers, each with different rules and requirements with respect to discounts, fees, and taxes. The program models this with subclasses dedicated to each type of customer; this allows for quicker access and computation of the financial details specific to each type of customer.

The application also incorporates a sorted list ADT, which holds an arbitrary number of customer instances and maintains an ordering based on either name, order total, or customer type.

In regards to data storage, the program utilizes a MySQL database, where a table is devoted to each piece of information (i.e. customers, orders, products, etc.). The application connects to and interacts with the SQL database using an API via JDBC API.

2.1 Alternative Design Options

Alternatively, the application could be constructed without OOP design principles. This might require less actual coding; however, the code would be very complicated and intricate. The approach of using objects seems much more suited to this order-invoice system.

Moreover, the data could be stored in some other manner besides a database, i.e. a spreadsheet or some other flat data file. However, using a database allows for better data integrity through validation of data and prevention of duplicate records. A database also allows for quick and easy creation of, retrieval of, updates to, and deletion of records. Since Jenn-Eric processes many orders and has a large number of products and customers, the use of a database to store the necessary data seems like the only suitable option for this order-invoice system.

3. Detailed Component Description

This section gives a more detailed description of the components summarized in Section 2. This section also gives a description of the testing methods use to test each component of the application.

3.1 Class/Entity Model - Database

The MySQL database schema is designed and implemented as Phase III of development of the Order-Invoice System. The database is designed to have the basic SQL CRUD functionality, as well as having as much data integrity as possible. In the database schema, tables are created for each of the major data types involved in the order-invoice system, i.e. customers, products, and orders. In order to obtain a many-to-many relation between the Orders table and the Products table, an OrderProducts join table is also created. Finally, specific tables are dedicated to smaller pieces of data involved in the Order-Invoice system, such as customer emails and physical addresses, in order to completely normalize the data.

The ER diagram given in **Figure 1** below details the database schema design.

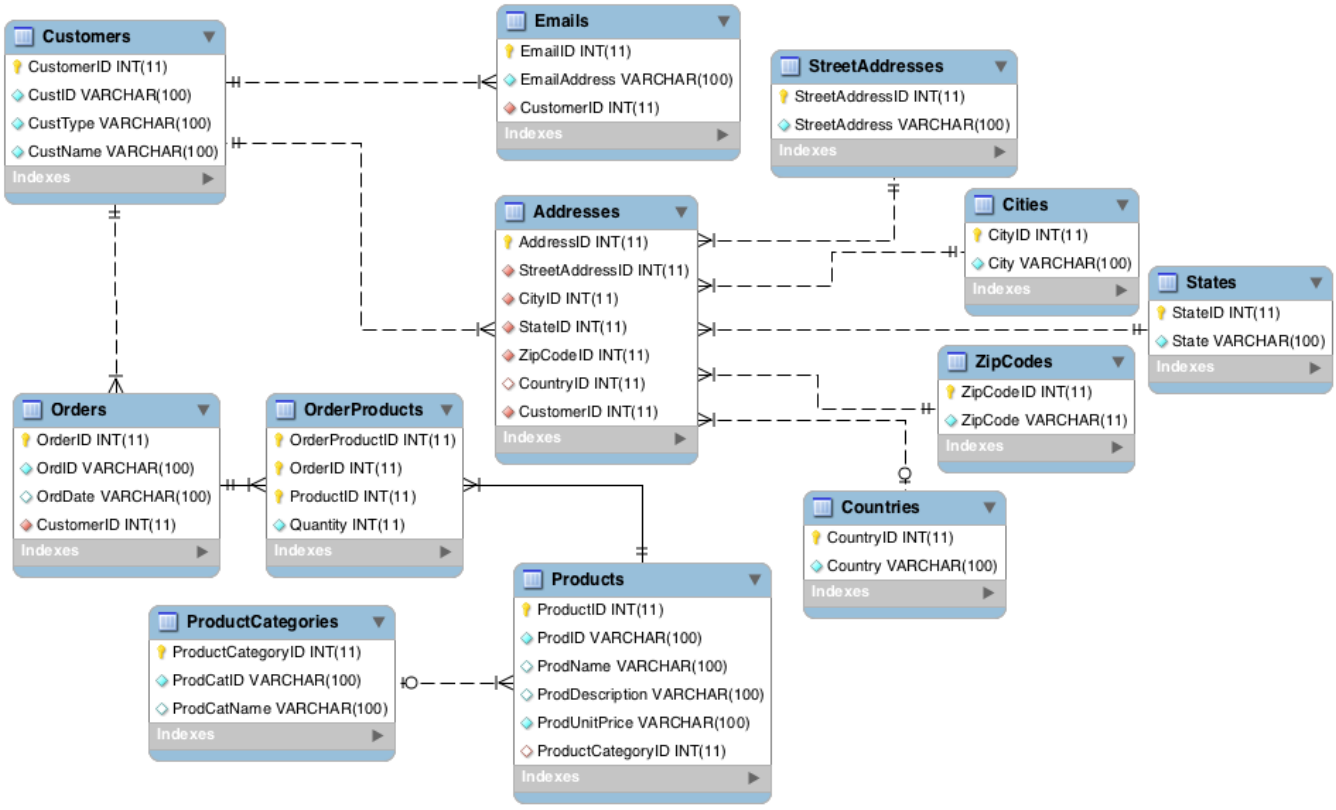

Figure 1. The Order-Invoice database

3.1.1 Component Testing Strategy

This component is tested by inserting all of the data from Phase I/II's test case into the Order-Invoice database (see Section 3.1.2 of this design document for details on the test case) and providing the database with several queries of varied types. More specifically, there are several test queries written to obtain a record or records from each of the tables. There is at least one test query to insert data into each table and at least one test query to remove data from each table. Finally, there are several test queries dedicated to updating tables. These queries attempt to emulate every possible request that could be made of the database once it is integrated into the current invoice printing system.

3.2 Class/Entity Model - Java application

The framework for the Java application is designed and implemented as Phases I and II of development of the Order-Invoice System. Classes are dedicated to each of the major data types involved in the Order-Invoice system, i.e. customers, products, product categories, and orders. Classes are also dedicated to smaller aspects of the Order-Invoice system to simplify the design and better follow OOP design principles.

The UML diagram given in **Figure 2** below gives an accurate description of the structure of the Java application.

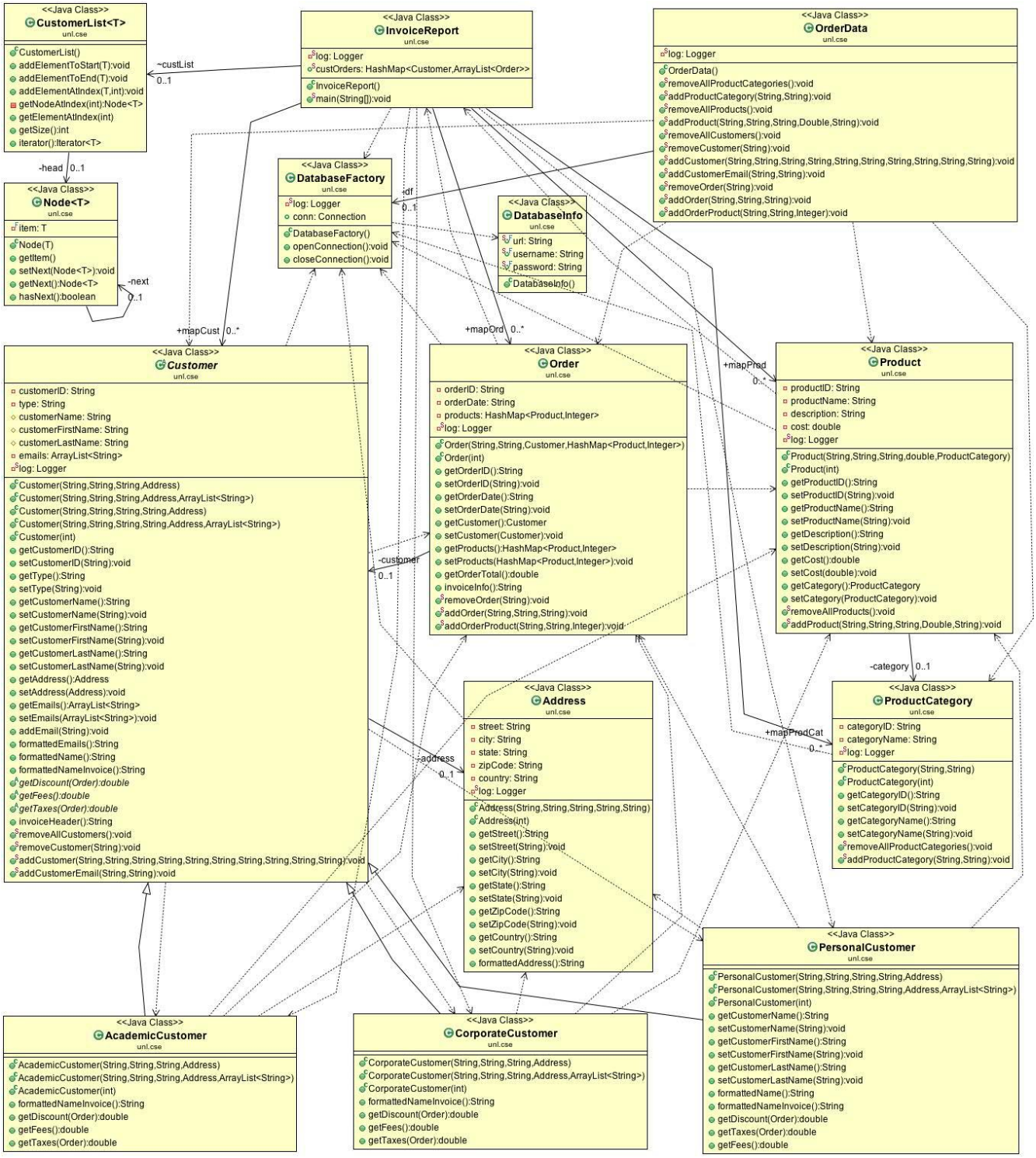

Figure 2. The invoice printing system

The design of the Java application realizes the four main OOP design principles in the following ways:

- **Encapsulation**

  Any and all data and functionality for each object is kept inside the class devoted to that object. For example, each class has a method which converts the data to a string appropriately formatted for an invoice report. Alternatively, this conversion to a string could be done inside the main InvoiceReport class but doing so would break encapsulation. Encapsulation is used further in the Java application in the fact that an object's state is not lost by manipulating the object and each class has getters and setters for every field to control access to the private members.

- **Abstraction**

  The classes used in the Java application are not simple data containers, but, rather, they have non-trivial methods. For example, the Address class does not simply hold address details; it also has a method to return a string with the address correctly formatted for invoice printing. Furthermore, the Customer class has several abstract methods that involve the financial aspects of an order- the fees, discounts, and taxes- since these aspects are specific to each type of customer. These abstract methods are, by necessity, implemented in each subclass of the Customer class. Finally, the use of any of the objects in the Java application does not require specific knowledge of the object's state or implementation and, therefore, properly making use of abstraction.

- **Inheritance**

  Inheritance in the Java application can be seen in the Customer class and its subclasses. There are several methods in the Customer class which are overridden in its subclasses. For example, the method in the Customer class which returns a formatted name is overridden in the PersonalCustomer subclass since a personal customer has a first and last name instead of a single name like an academic or corporate customer.

- **Polymorphism**

  The Java application has several instances of the use of polymorphism, but the main example would be the constructors in the Customer class. While academic and corporate customers have just one name, personal customers have a first and last name. Therefore, a different constructor is used to create a customer with only one name than to create a customer with both a first and last name. Similarly, it is possible for a customer to have an email address (or multiple emails) or no email address, and the application models this with separate constructors for each case. Of course, Java automatically knows which constructor to call because the fields of each one are different.

3.2.1 Component Testing Strategy

This component is tested by providing the program with a test case. The test case contains the four input data files: Customers.dat, Orders.dat, Products.dat, ProductCategories.dat (which are flat data files, i.e. are not read from a database), as well as the expected output, in a .txt file. The test case contains customers of all three types: academic, corporate, and personal, to test the functionality of each of the subclasses and the overridden and/or abstract methods. The test case also contains customers with no emails, one email, and multiple emails to test the constructors in the Customer class. Finally, the test case contains customers who placed no orders, one order and two orders to test the ability of the invoice printing function to create instances of every class and relate those instances. Certainly, the test cases attempt to reach every branch of the program, and because there are cases created to specifically test certain aspects of the application, when the actual output is compared to the expected output, it is easy to see where, if anywhere, the program is failing.

3.3 Database Interface

The database is integrated into the application with the use of an API via JDBC API as Phase IV of the development of the order-invoice system. Records are loaded from the database and into Java objects by connecting to the database, selecting specific data records via queries to the database, which creates a result set of data, and finally creating Java objects from the result set. The API persists data to the database by again connecting to the database and then updating tables and/or inserting data via update queries. Of course, with any SQL database, deletion of a particular record may not occur in just one table, since many tables are connected using foreign keys. That is, if data is to be deleted, it must be deleted from every record in the database in the proper order. For example, suppose a product record is to be deleted; then any entries in the OrderProducts table which reference the specified product must first be deleted before the specific product record can be deleted from the Products table.

The Java application contains an OrderData class which has methods to add and remove records of every type. To maintain proper encapsulation, however, these methods are actually implemented in the individual classes to which the methods relate, and then called from the OrderData class. The Customer, Order, Product, and ProductCategory classes also have specific methods to retrieve records from the database and construct instances of the respective objects based on the data obtained.

Data validation occurs at both the database and code level. To avoid duplicate records, certain columns of each table require unique entries (i.e. database level), but in the Java application, before inserting a record into the database, the application first queries the database to see if that record already exists (i.e. code level). A specific example of this can be seen with respect to customer records. The database requires a unique external customer ID (i.e. the alpha-numeric customer ID given to each customer by Jenn-Eric), so any attempt to insert a duplicate customer record with the same external customer ID is not successful. However, as further validation, the Java application first checks to see if such a customer record exists before trying to insert a new one.

Instead of being silenced or sent to the standard output, all errors (including SQL errors/exceptions) which occur while using the API are logged (i.e. output to system log files) using the logging system Log4j.

3.3.1 Component Testing Strategy

This component is tested using the same test data as in Phase I/II of the design (see Section 3.2.1 of this design document). The data is loaded into the database in Phase III of the design (see Section 3.1.1 of this design document). The application now reads the data from the database (vs. reading from flat files previously) and prints invoices and reports similar to Phase II of the design, so the actual output of the application is compared to the same expected output given in Phase II (see Section 3.2.1 again). Since errors are logged and output, it is easy to see when an error has occurred due to erroneous code and/or incorrect SQL queries or updates.

3.4 Design & Integration of Data Structures

The final phase (Phase V) of the development of the order-invoice system is the design and implementation of a sorted list ADT. The list ADT can hold an arbitrary number of generic objects, but in the context of this application, is used only to hold Customer objects. It maintains an ordering of the customers based on certain criteria, including the support of the following three orders: 1) ordering customers alphabetically by name (company name for academic and corporate customers and last then first for personal customers), 2) ordering customers by highest-to-lowest order totals, and 3) grouping by customer type and then ordering customers alpha-numerically by customer ID.

The list ADT is link-list based, so the list only keeps track of the head node of the list and then each node points to the next. As such, resizing the list when adding or deleting elements is unnecessary. Additions to the list are

addressed by starting at the head node, moving from node to node until the proper spot is reached, linking the previous node to the new node and then linking the new node to the node which was originally pointed to by the previous node. Similarly, removals from the list are achieved by again starting at the head, moving from node to node until the proper spot is reached, deleting the specific node, and then linking the previous node to the node immediately following the deleted node.

The list is constructed by creating an instance of the CustomerList class and then adding elements by using the methods in the Node class. To unify the design, the list is constructed along with a Comparator which the ADT then uses to maintain its ordering. Also, the ADT implements the Iterable interface in order to be able to be iterated over and usable in an enhanced for-loop.

3.4.1 Component Testing Strategy

This component is tested using the same testing strategy as in Phase IV (see section 3.3.1 of this design document). However, instead of creating a customer HashMap, the InvoiceReport main function now creates a CustomerList of customers and prints the same summary report as in Phase I/II three different times where each time, customers are ordered in each of the three possible ways given above. Any failures or errors in the list ADT are readily apparent if any customers are missing from the summary report or are in the wrong order on the summary report, and the list ADT is adjusted accordingly.

3.5 Changes & Refactoring

In Phase II of the design, methods were added to the Customer and Order classes to output certain order information necessary for an invoice. The purpose of this change was to further encapsulate related data and methods in their respective classes.

As Phase III of the design was entirely focused on database design, no changes were made to the invoice printing system.

In Phase IV of the design, tables were added to the database to normalize geographical data. Also, the database was updated to create a one-to-many relationship between the customer and emails table, instead of the many-to-many relationship, which was created previously. Finally, duplicate Order records were further prevented by placing a unique keyword on the combination of OrderID and ProductID in the OrderProducts table.

In Phase V of the design, a OrderProductID column was added to the OrderProducts table for consistency. This column now serves as a primary key for the OrderProducts table.

4. Bibliography

1. Arapidis, C. (2012, September 25). *How to generate UML diagrams from Java code in Eclipse*. Retrieved February 13, 2013, from http://fuzz-box.blogspot.com/2012/09/how-to-generate-uml-diagrams-from-java.html
2. Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.
3. *Entity-relationship model (diagram)*. (2013). Retrieved from February 27, 2013 http://www.webopedia.com/TERM/E/entity_relationship_diagram.html
4. *Java platform, standard edition 6 api specification*. (2011). Retrieved February 13, 2013, from http://docs.oracle.com/javase/6/docs/api/
5. *UML: Unified modeling language*. (2002, January 01). Retrieved February 14, 2013 from http://foldoc.org/UML
6. *XML*. (2013, February 13). Retrieved February 14, 2013 from http://en.wikipedia.org/wiki/X