DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

Invoice-Order System

CSCE 156 – Computer Science II Project

Student 002

4/4/2013

Version 1.4

The descriptions and content in this document describe the designs and aspects for a project that uses the object oriented paradigm backed by a database for an order-invoice system.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Student 002	2013/02/01
1.1	Second Draft (Added Class Sections), Basic Revisions	Student 002	2013/02/15
1.2	Third Draft (Added Database Sections), More Revisions	Student 002	2013/02/28
1.3	Fourth Draft (Added Database Connectivity), Revisions, Added ER and UML Diagrams	Student 002	2013/03/14
1.4	Fifth Draft (Added ADT Section), Finalized Revisions and Sections, Updated Diagrams	Student 002	2013/04/04

Contents

Introduction

This document outlines the different components and designs for an API that models an order invoice system. The API uses an OOP and Database paradigm to accomplish the necessary tasks given by the project's scope.

1.1 Purpose of this Document

The document itself serves as the blueprint to implement the order-invoice API and details the multiple elements of the design. This document is meant to convey the ideas of the inner workings of the system and how the elements within the system interact with each other to process the data given to the API by the user. The document only provides the necessary specifications for the implementation of the defined API.

1.2 Scope of the Project

The API models the order invoice system, so it processes the orders for customers and outputs their individual invoices. Simply, a customer places an order and an invoice is produced for that order. The invoice itself represents the given data by the order.

The details of each invoice are as follows:

- Values for the customer and order details are listed
- The products with their quantities along with their line item totals are calculated
- Any applicable per item then per order discounts are calculated for a subtotal
- Any applicable service fees are added and discounts subtracted for another subtotal
- Sales tax (if any) is calculated and added for the grand total

Each invoice follows this logic to produce an order invoice for that customer. Any discounts, service fees and sales tax values depend on the business logic used for the design.

This API also produces a summary report for the invoice order system that includes the combined totals for the invoices for each customer. The API doesn't keep track of these individual invoices or summary reports thus they must be reprocessed when needed, though all class data should be available in the records of the database.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Category: Grouping of unique products types.

Comparator: A comparison function, which imposes a *total ordering* on some collection of objects. [1]

Customer: the client that the company serves. The company serves three types of clients: personal, corporate, and academic.

Invoice: A detailed report that represents the list of products that the customers buy along with added costs and totals.

OOP: A paradigm that uses classes that creates instances of objects with unique fields and methods.

Order: The entity placed by customers to purchase different products.

Product: The item that the company sells to their clients.

XStream: A simple java library to serialize objects to XML and back again. [2]

1.3.2 Abbreviations & Acronyms

ADT: Abstract Data Type

API: Application Programming Interface

ER Diagram: Entity-Relationship Model

JDBC: Java Database Connectivity

OOP: Object-Oriented Program

SQL: Structured Query Language

UML: Unified Modeling Language

XML: Extensible Markup Language

2. Overall Design Description

Since this project uses the OOP paradigm it's broken down into several unique classes. [3] Each individual class possesses its own fields for relevant data and methods to allow for the functionality of the class. The classes each represent the different aspects of the project. There is a class for the customer, personal, academic, corporate, product, category, address, and order entities. Each class is designed for abstraction to allow for the reusability and for independent status in the API.

The personal, academic, and corporate entities are sub classed to the customer class where the customer is an abstract class, because they possess an 'is-a' relationship. The subclasses inherit the traits of the customer and act like an instance of that customer. The addresses class belongs to an instance of the customer class though it doesn't inherit any traits like the subclasses.

The customer class is in charge of the actual process of producing the invoice. In this project the order and product class are mainly data containers while the customer class does most of the brute force work. This design follows that a customer 'has-an' invoice, so the design lets the customer class handle this procedure.

2.1 Alternative Design Options

Other design options in this project considered in development include the following:

- One design would involve personal, academic, and corporate not being subclasses, but just regular classes. This design works, but it doesn't take advantage of the inheritance that comes from being sub-classed and it's simpler to declare a sub-classed customer type as an instance of a customer as each of the unique instances doesn’t have to be handled differently.
- The address class could be omitted and the address fields handled by the customer class.
- Another design involves letting the order class handle the invoices as opposed to the present design. There is a 'has-a' relationship between the orders and invoices like with the customer, so this design makes sense, but then order invoices would be handled individually per order rather than invoices of the customer. Either design method works out similarly in this case, so there isn't a significant advantage over either choice.

3. Detailed Component Description

The API uses classes to represent the instances of these entities given by the data structure. These instances are created using constructors along with given data values. These data values are paired with the classes and now belong to that class. These fields are encapsulated, which allows them to be private states that cannot be accessed by the outside world directly, except by their parent class.

3.1 Class/Entity Model

The ER Diagram in figure 1 represents the database schema of the order-invoice API. The diagram displays the fields and relations of the several tables representing the class entities of the API.

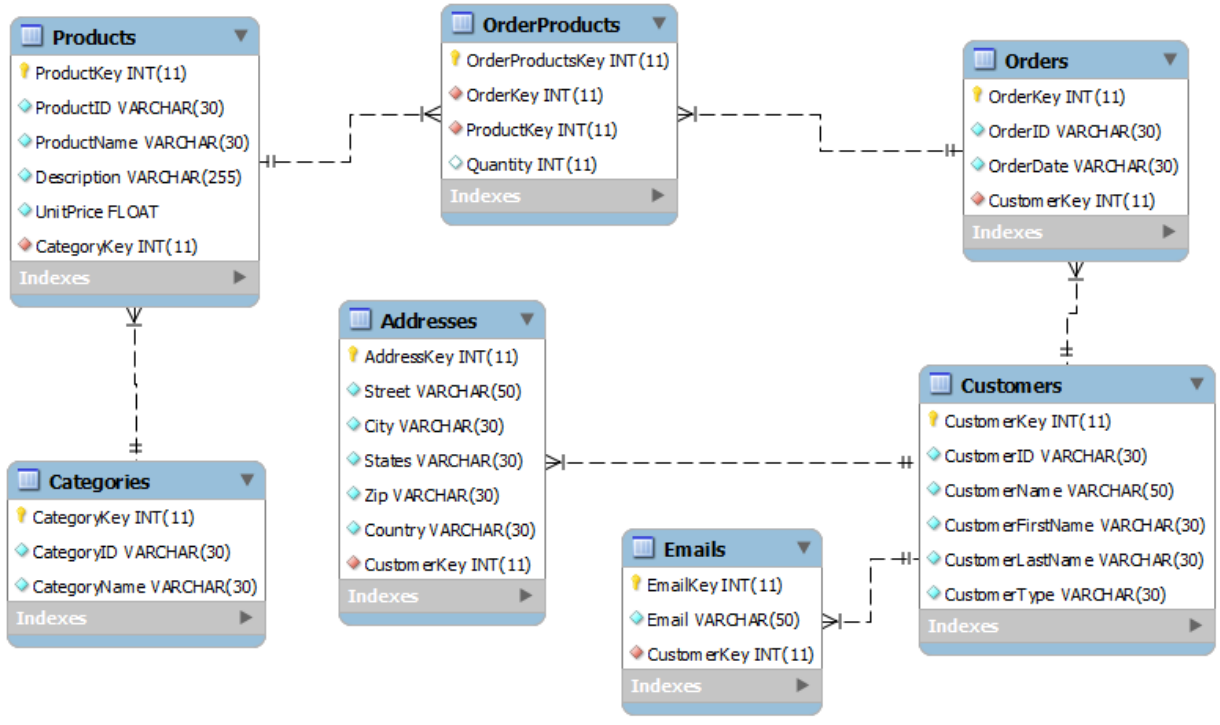

Figure 1 An ER Diagram representing the tables and their fields

3.1.1 Component Testing Strategy

The database is tested through a sample of test of SQL queries to determine if the data is stored and retrieved correctly.

3.2 Class/Entity Model

The UML diagram in figure 2 represents the state and functionality of the classes in the API.

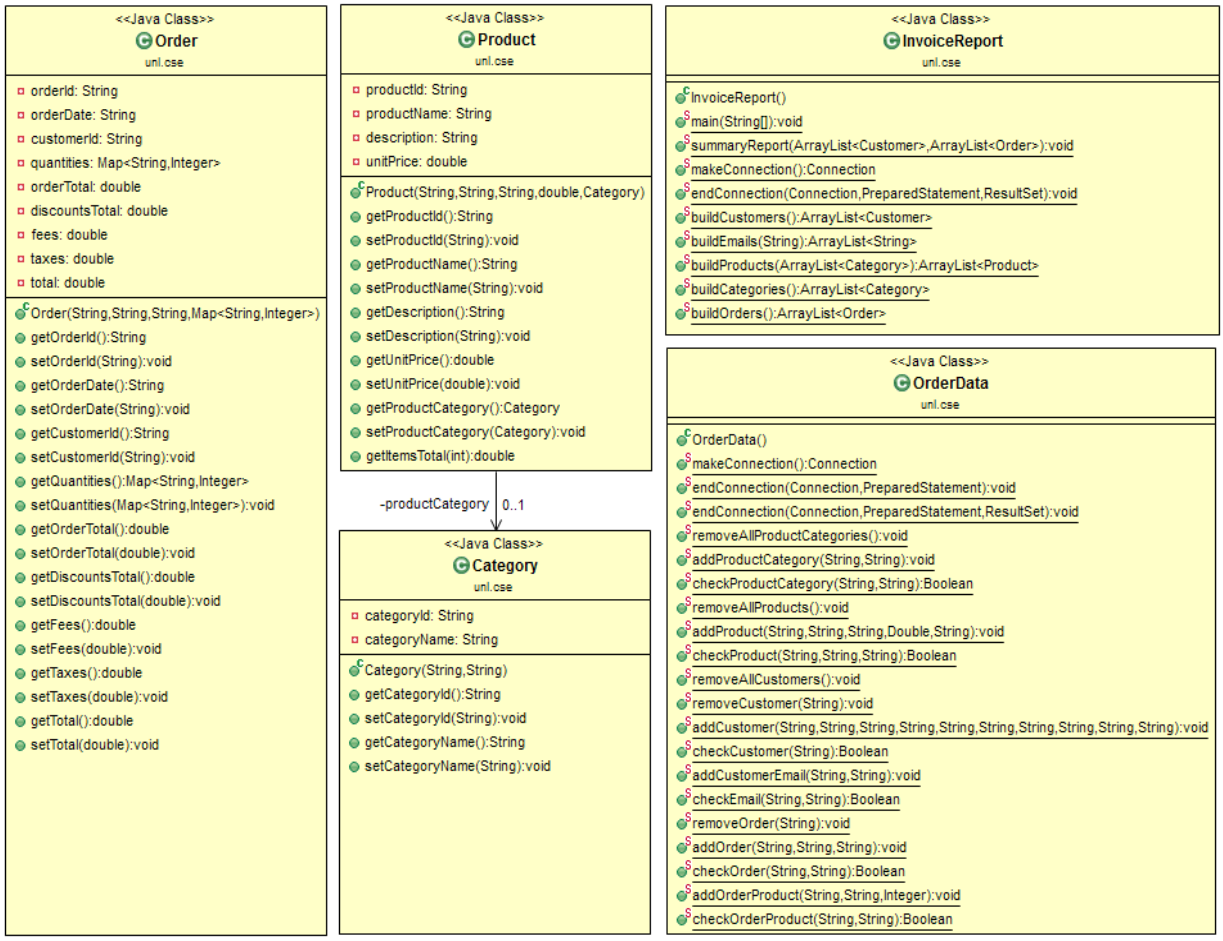

Figure 2 An UML Diagram displaying the classes and their relations

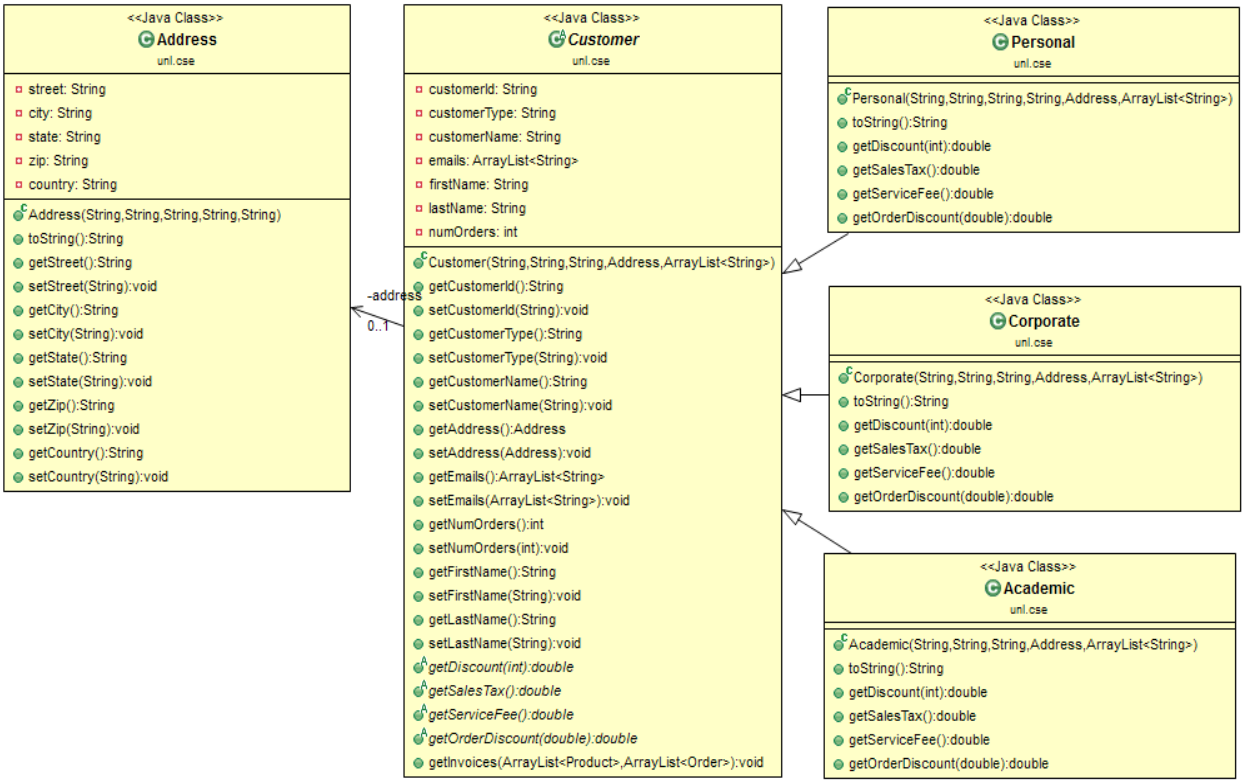

Figure 3 An ER Diagram displaying the classes and their relations

3.2.1 Component Testing Strategy

The fields and relations of this design were tested through the creation of a XML data file. A file was created for both the customer and the product entities. The file contained XML code blocks that represented the data values for each field obtained from a data file. A separate class created instances of these objects with the test data and output them to the XML file using the XStream library to convert the java classes to XML. Each block successfully contained the necessary data fields and values correctly, so the test succeeded in demonstrating that the classes data structure operated correctly.

The interaction of these classes was tested through a test in processing the invoices for customers from test data. A separate class was created again for this purpose which created instances of the entities and used the invoice method of the customer to output the invoices. The invoice uses the instances of the products and orders to build the invoice, so it tested the interaction between these objects, so that all the data from these classes produced the correct invoice according to the scope of the project. Also, a summary report was produced along with the invoices.

3.3 Database Interface

To include the addition of database connectivity the API includes a new class to handle interactions with the database. The new driver class uses the MYSQL driver with JDBC to make connections with the database schema to retrieve, update, or delete records. JDBC allows the API to make this connection to the database, query for the data, and obtain results. [4] The records are attributed to the classes in the API by querying the database for the desired data records.

In order to maintain the data integrity of the database, some records are destroyed along with others. When a customer is removed, all orders and emails associated with that customer are also deleted. Products and addresses must remain because they are independent of the customer.

Also, to maintain the integrity of the database, some fields inserted into the database are validated for correctness. The API checks against the customer type, and products ordered. The customer type can only relate to the types of customers the API handles, so it must be restricted to those types. The products the customer purchases must exist in the database in order to be bought, as the API can only handle products it knows.

The API handles null data by rejecting certain null fields. A customer does not accept null data on the customer type, as the API can only handle specific types as mentioned earlier. Also, the customer name should not be null, as each order should have a name. Other fields, like the email, address, and orders for that customer are allowed to be null by the API because they are optional data entries by the user.

3.3.1 Component Testing Strategy

The interaction between the API and the database was tested by updating the driver class from the earlier test to process database records rather than flat files. The class used builder methods that connected to the database integrated for this API as detailed in the Database Interface section. It used the resulting data records to create the class instances and then processed the invoices and summary report accordingly.

3.4 Design & Integration of Data Structures

The ADT for this API is implemented through an array based list. The array list holds the customer objects in an order designated by the called comparator for the list. The comparator method added to this ADT structure maintains the order of the list as objects are added and removed during its lifespan. The list possesses no internal sorting methods of its own and relies on the comparator method for its ordering.

Since the list is array based it carries a static capacity that must be accounted for when elements are removed or added to the list. To handle these situations, an expand method copies the elements over to a new array of proper size to carry the new element if the current size is too small. When an element is removed then the size field of the array list can be decremented for the current capacity. When an element is added it must be added according to the current ordering of the list. It must run through the list with the comparator method to find out where it belongs. Other necessary methods like finding if the array is empty or finding element at an index are also added to build the lists functionality.

The implementation of this list is general and follows parameterized polymorphism where any object can be added to the list and maintaining functionality.

3.4.1 Component Testing Strategy

The ADT was tested by building three different instances of the lists for several orderings. The list orders follow descending alphabetic ordering of last name / first name, total of orders from highest to lowest and an ordering that groups the different customer types and groups them by their customer ids

descending in an alpha-numeric order. A comparator method was created to build the ordering of the list and maintain that ordering as elements were added to the list. These lists tested the functionality of the array list structure and its methods and printed out all the orderings. Since the comparator maintained the ordering throughout the addition and removal of elements the implemented ADT operated correctly.

3.5 Changes & Refactoring

During the development of the API, some necessary changes were needed to be made in order to adapt the API to the current data inputs or adjust present values to maintain the database's integrity. To incorporate the database connectivity the driver needed to be changed for the different data inputs. The data type for emails was changed from the array of strings to an array list, so emails could be added using the methods available in the collection library rather than indexed to a value directly.

In the database, multiple changes were made to improve how the data was handled. The productCategories JOIN table was dropped as it was deemed unnecessary to the schema. Also external Ids were added to some of the tables and character sizes were increased to adjust for larger inputs. The order table's numerical values for the order amount totals were removed as those values are calculated and handled in the API and not needed in the database itself.

4. Bibliography

[1] Mkyong. (2010, July 7). *Java object sorting example (Comparable and Comparator)*. Retrieved April 4, 2013, from Mykong.com: http://www.mkyong.com/java/java-object-sorting-example-comparable-and-comparator/

[2] *About XStream*. (2013, January 19). Retrieved March 14, 2013, from XStream: http://xstream.codehaus.org/

[3] *OOPs – Object Oriented Programming System*. (2008, August 24). Retrieved April 04, 2013, from Basic OOPs Concepts : http://oopsconcepts.blogspot.com/

[4] *JDBC Overview*. (2013, March 14). Retrieved March 14, 2013, from Oracle: http://www.oracle.com/technetwork/java/overview-141217.html