Spring 2021

Sales System Design

Computer Science II Project

Joel Bargen and Noah Zetocha
UNL DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
VERSION 1.5

Abstract

Modern Geek Games is in need of a technology overhaul. The company was recently purchased and the previous owner had no concrete sales system. The goal is to make a complete sales system using Java that is database—backed. In order to handle the current needs as well as bringing the previous information up to date the company will need to have file processing and reporting capabilities.

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	Joel Bargen and Noah Zetocha	2021/02/16
1.1	Revision set 1	Joel Bargen and Noah Zetocha	2021/02/26
1.2	Revision set 2	Joel Bargen and Noah Zetocha	2021/03/16
1.3	Revision set 3	Joel Bargen and Noah Zetocha	2021/03/29
1.4	Revision set 4	Joel Bargen and Noah Zetocha	2021/04/14
1.5	Revision set 5	Joel Bargen and Noah Zetocha	2021/04/22

Contents

1. Introduction

This document outlines the design of a new sales system written in Java. The system is efficient, database-backed and built to support Modern Geek Games (MGG) sales system. Minerva Campbell, a local investor, recently bought the regional chain of MGG stores. MGG sells a variety of new and used game products as well as gift cards, subscriptions and accessories. They also provide varied services to their clients. The customer structure of the MGG store supports discounts for employees and a level of discount depending on the type of customer. The previous owner had a de-centralized system of macros and excel spreadsheets to keep track of data, which was not very technologically advanced. In order to improve the technological infrastructure of the store chain, the data had to be moved into a modern database, and the business logic for store sale functions was built. It was imperative to parse CSV files from excel and load the data into cross-platform xml format for greater flexibility. In addition, the data needs to be reportable in the case of sales, so reporting methods were implemented as well.

1.1 Purpose of this Document

The purpose of this technical document is to describe the construction of the system and outline the various objects in the system. The main purpose of this system is to -

- Provide a link between the specifications needed by MGG and the detailed design.
- Lay out the functionality that each component or group provides and show their interaction.

This document does not address installation or machine details of the implementation.

1.2 Scope of the Project

The new owner of MGG is updating the way sales records are maintained. To replace the previous owner's system of macros and excel spreadsheets, we designed a system that is database-backed and handles the store's different types of sales items. These items include:

- Items
- Persons
- Stores

More details on these classes can be found in the Overall Design Description. The design follows a three-tier architecture for the implementation and this project focuses on tier two and three, the application and database layers respectively. The presentation layer, or first tier is out of scope for this project since we aren't designing a graphical user interface (GUI) or web interface.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

1.3.2 Abbreviations & Acronyms

MGG -- Modern Geek Games

SQL – Standardized Query Language

CSV – Comma Separated Value

XML – Extensible Markup Language

JDBC – Java Database Connectivity

GUI – Graphical User Interface

ADT – Abstract Data Type

2. Overall Design Description

The system is designed in Eclipse using the Java language with SQL database functionality, and it will include structures that hold different sales items with file processing and reporting functions. Products of the store will include toys, apparel, gift cards, and games. Services will include services completed by MGG staff. Subscriptions for online services, licenses, or games will also be included in the sales system. So far, the design includes structures for items, persons, and stores that are associated with MGG along with the sale data for each sale occurrence. Included in the system are functions that process CSV files, parse them, and output XML files for cross-system use. The need for this file processing function arises from the previous owner keeping data in spreadsheets. XStream was helpful for both XML outputting and aliasing for improvement of the actual output. For system users to more easily access reports, it became important to include classes that took list structures of information, loaded from the csv files and compile them to a user-friendly output. Reports made were sales made by sales person, sales made at each store, and a detailed list of all sales made. By using lists within the employee, customer and store classes, of sales made, it was possible to sort by each depending on the requirements of the report. A sorted list ADT was built to improve upon the design of the sale lists. The ADT will be discussed in a later section. To make the design more functional data parsing and the business logic are handled separately. This allowed for an easy switch to a database for storing data, as discussed later.

2.1 Alternative Design Options

The current system arose from recognizing the need for a hierarchy in the person and item objects, with some objects sharing information. Originally each object had its own extended sale class. As necessary for better functionality for sorting it may be possible to change each sale instance in the other classes to a list of sales, affording greater searching and sorting capabilities. Adding in the sale functionality was challenging at first but inheritance proved to be useful in extending sale classes to the items. Originally it had another inherited regular customer that was removed in lieu of the salesperson class. Extending sale classes to the items proved to be the best option compared to the previous design. The original draft of the database design had the address object in one table. This design, while relatively simple in execution, was not the most efficient design. Instead, the address table was split into its respective parts, and individual tables then held states and countries, preventing repeated information being stored.

3. Detailed Component Description

The software has database functionality and is built with a multi-class framework for disseminating data.

3.1 Database Design

MySQL is the software used to build the database. It is connected to the server used for this program. Each object has a table built with the same structure that the objects in Java have. The tables included are items, persons, stores, addresses and sales. Country and State tables are included to reference in the address table. Each table has primary keys that represent their unique SQL identifier keys and foreign keys linking the tables. The store and person objects will be linked by address and sale. The items will be linked to the person making and receiving the sale. Sale and item classes will be linked with the SaleItem table, providing more detailed sale information from the database. Likewise the Email table will be linked to the Person table by a PersonEmail table, facilitating a multiple email relationship to a person. The decision to build the database this way arose from the need for a single sale to have multiple sale items linked to it at the same time, without replicating the item or the sale itself. For the same reason we made the person email table to prevent re-making emails and the person. This will provide a good bridge between the current Java only framework and one that connects and queries the database. This database design will ensure that the initial problem of a lack of data uniformity and access is addressed.

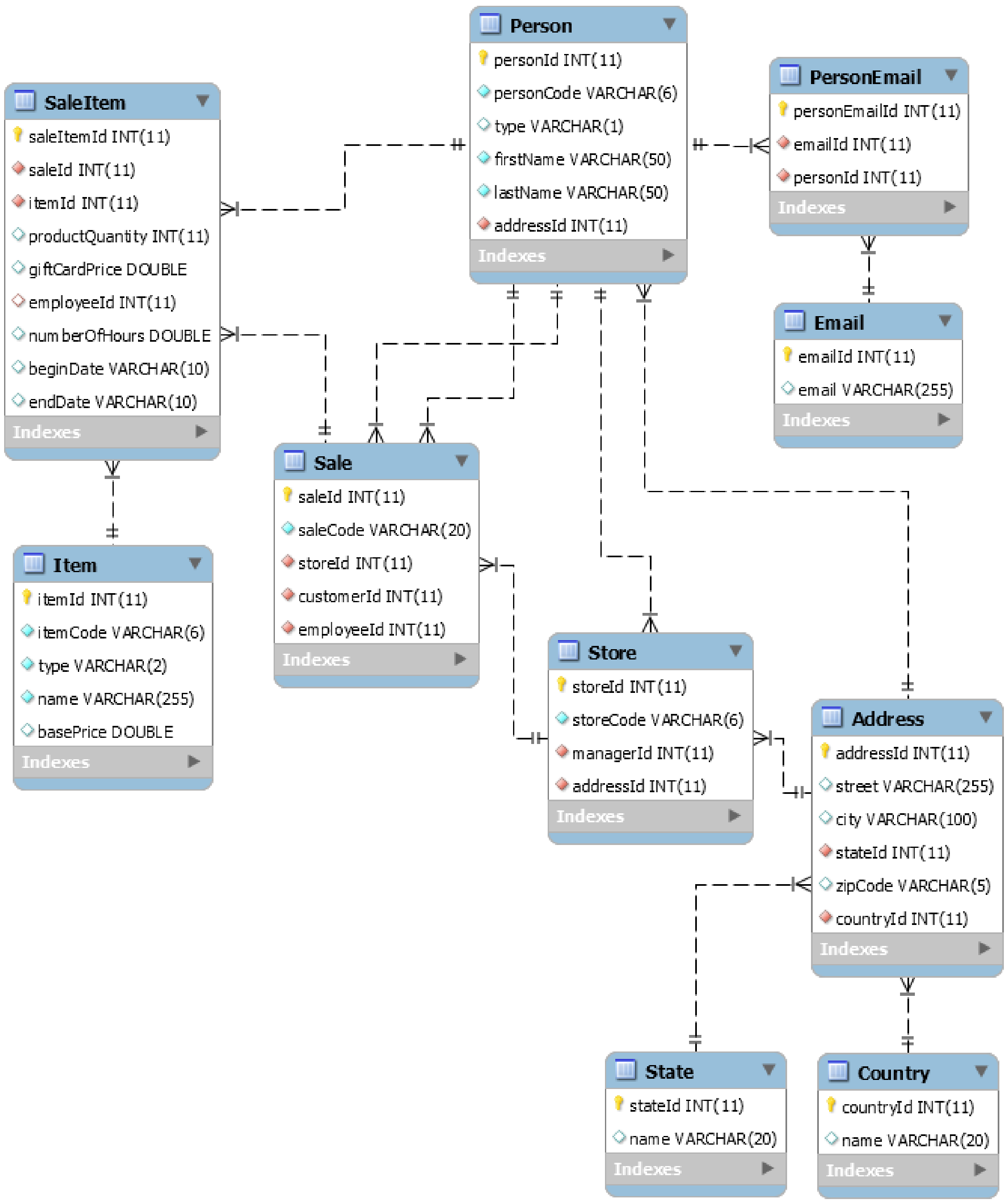


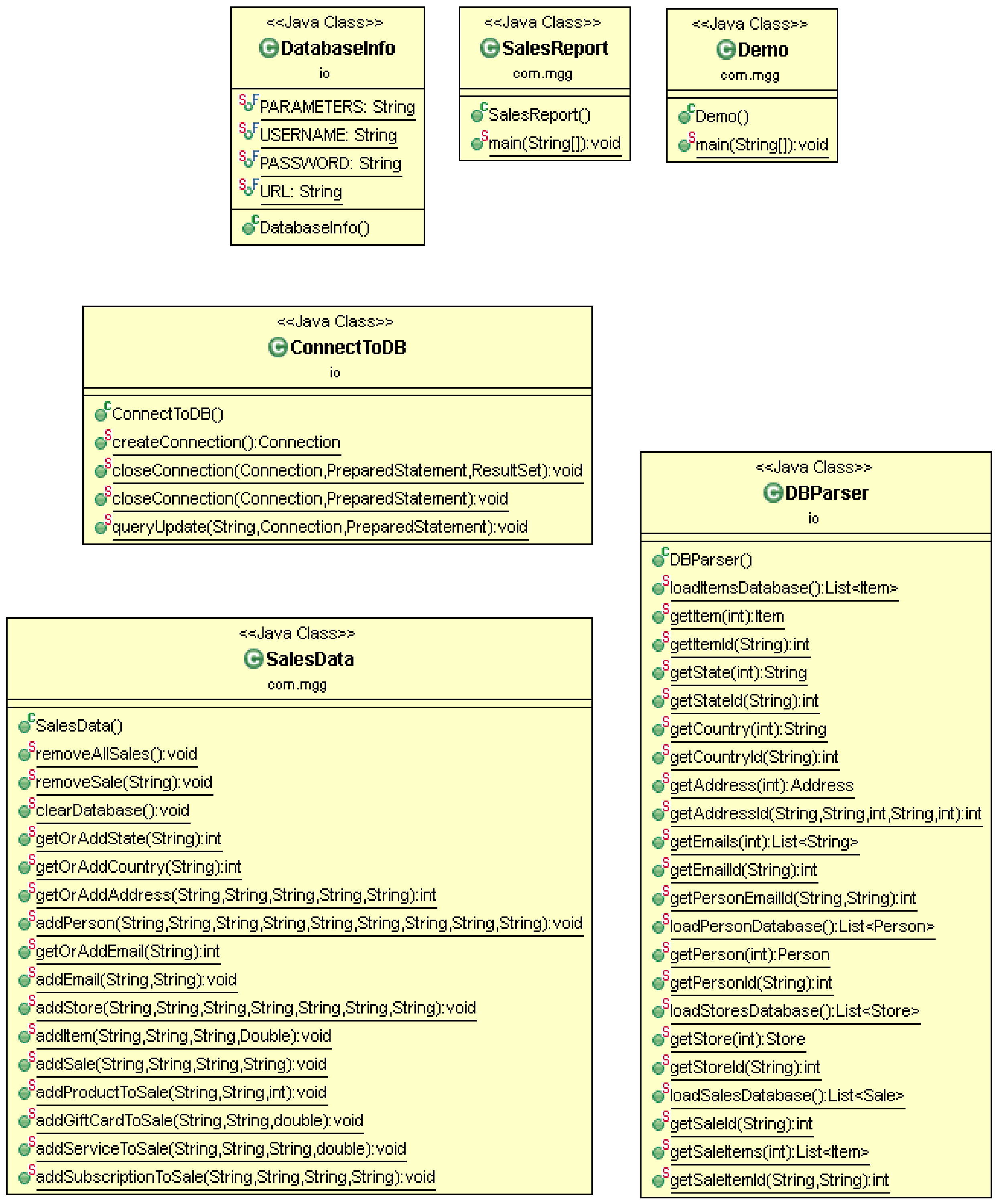
Figure 1: The ER Diagram representing the table structure and relationships used in the construction of the database for MGG. Image produced using MySQL.

3.1.1 Component Testing Strategy

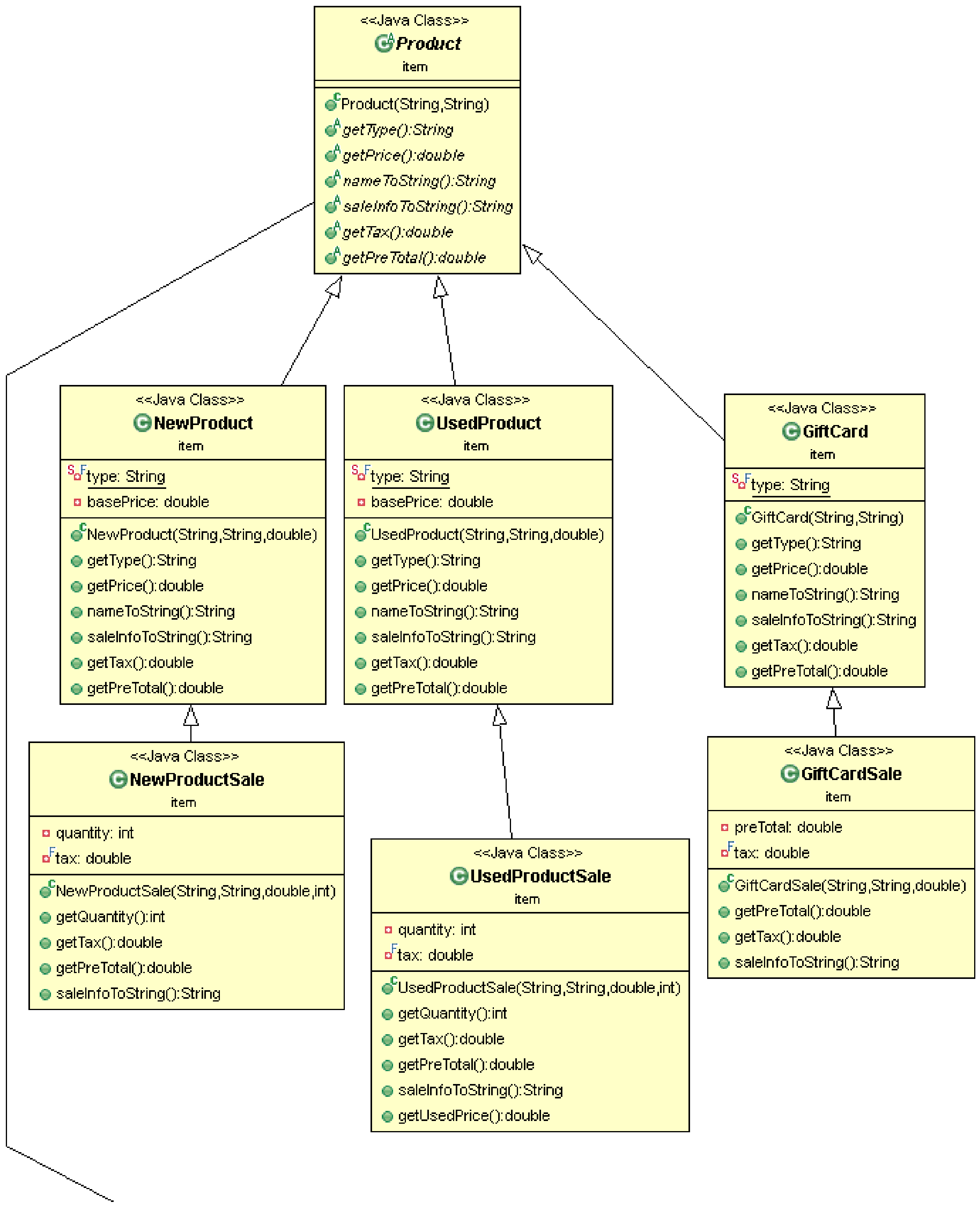
The database in MySQL is tested using verified testing data in order to show that proper insertion of data into the database works and that it allows runnable queries with predictable output. Verification can be done using the standard output within MySQL. Data for all table types are inserted and the relationships between them are tested to ensure proper working relationships within the database. In order to test a complete sale, 5 test sales were created, with 1 to 3 sale items each, and these linked to the 12 items made. For these sales, 8 stores were built and 20 address made for the people or store. To build the people, 34 emails and person email types were included to make the 12 people associated with the stores and sales. With this testing framework there were generally good results for the tests. Given the final design, the tests passed with a variety of queries.

3.2 Class/Entity Model

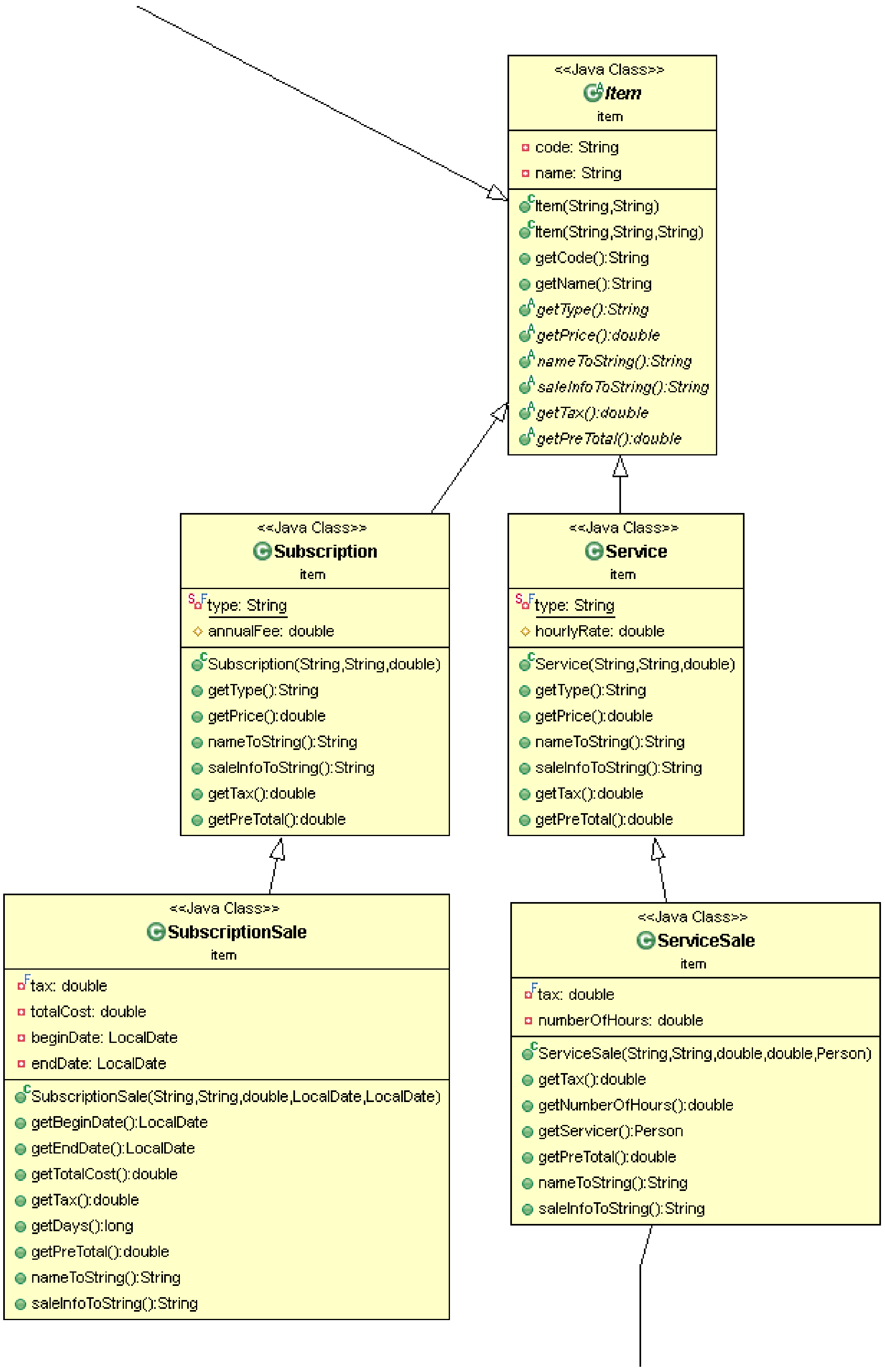
The design includes classes for items, stores, addresses and persons and sales, with functionality for each data object in exterior CSV parsing and XML outputting classes. The functions are run using the data converter class. The item class has subclasses including classes for new and used products, gift cards, subscriptions, and services. The person class has two subclasses, one for employees and one for customers. The employee subclass has subclasses built for managers, salespeople and regular employees. The customer subclass has subclasses built for gold customers, platinum customers and regular customers. The store class is built to represent the various stores owned by MGG and the people associated with the store. Unique store codes and associated people codes are used. In addition to these classes, we also have an address class that is tied to both the person and store classes. By using a unique person code the design allows for the association between store and person with further association by other classes possible, based on unique codes built into the system or parsed from existing company data. The sale class defines sales by unique identifier codes, the person who made the sale (salesperson) and the person who bought the item. Each item then extends a sale version of the item for data reporting. Proper rounding was built into the classes to facilitate dollar and cents amounts rather than doing math with unrounded doubles.



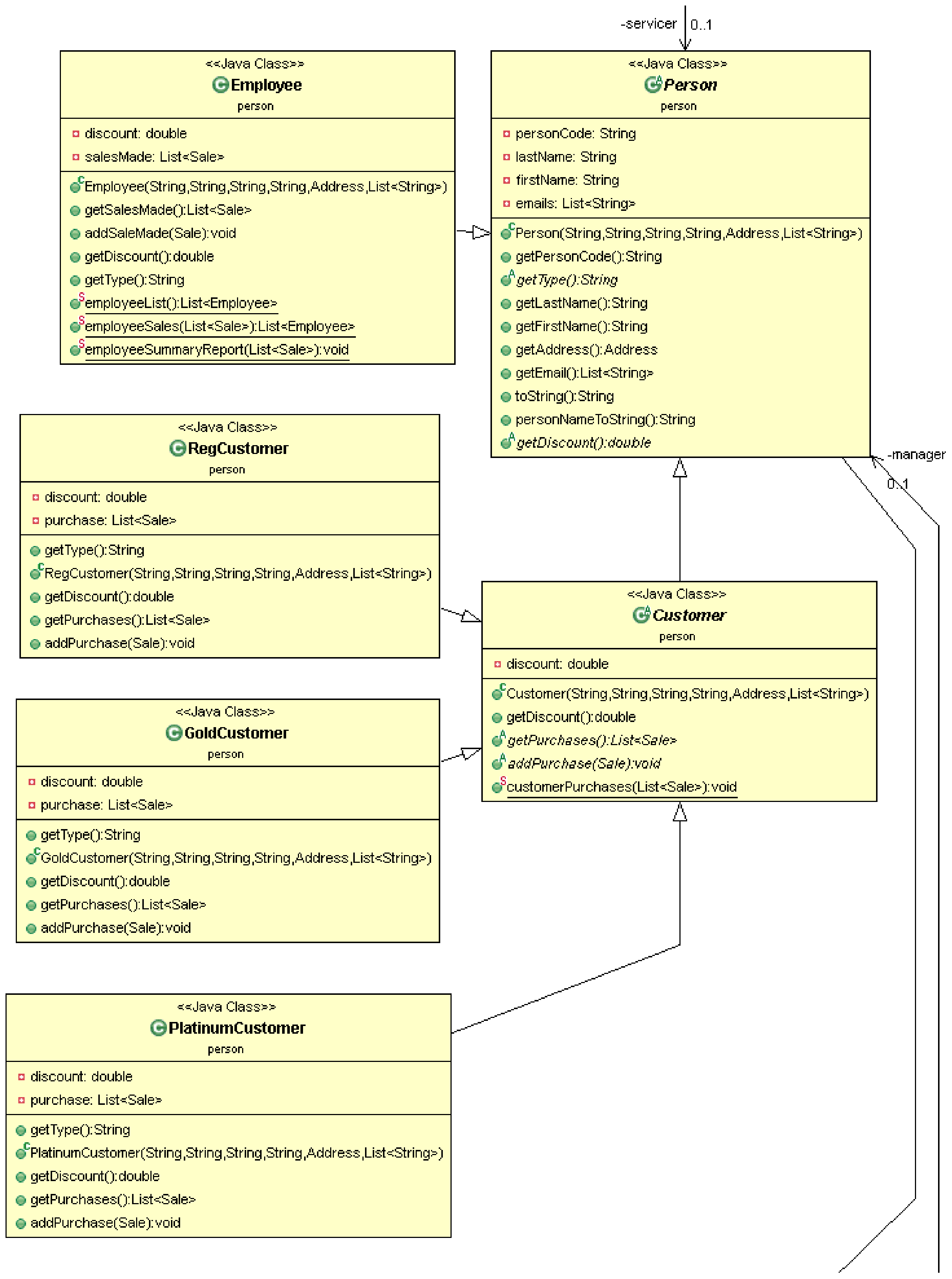
<<Java Class>>
DatabaseInfo
io
PARAMETERS: String
USERNAME: String
PASSWORD: String
URL: String
DatabaseInfo()
<<Java Class>>
SalesReport
com.mgg
SalesReport()
main(String[]):void
<<Java Class>>
Demo
com.mgg
Demo()
main(String[]):void
<<Java Class>>
ConnectToDB
io
ConnectToDB()
createConnection():Connection
closeConnection(Connection,PreparedStatement,ResultSet):void
closeConnection(Connection,PreparedStatement):void
queryUpdate(String,Connection,PreparedStatement):void
<<Java Class>>
DBParser
io
DBParser()
loadItemsDatabase():List<Item>
getItem(int):Item
getItemId(String):int
getState(int):String
getStateId(String):int
getCountry(int):String
getCountryId(String):int
getAddress(int):Address
getAddressId(String,String,int,String,int):int
getEmails(int):List<String>
getEmailId(String):int
getPersonEmailId(String,String):int
loadPersonDatabase():List<Person>
getPerson(int):Person
getPersonId(String):int
loadStoresDatabase():List<Store>
getStore(int):Store
getStoreId(String):int
loadSalesDatabase():List<Sale>
getSaleId(String):int
getSaleItems(int):List<Item>
getSaleItemId(String,String):int
<<Java Class>>
SalesData
com.mgg
SalesData()
removeAllSales():void
removeSale(String):void
clearDatabase():void
getOrAddState(String):int
getOrAddCountry(String):int
getOrAddAddress(String,String,String,String,String):int
addPerson(String,String,String,String,String,String,String,String,String):void
getOrAddEmail(String):int
addEmail(String,String):void
addStore(String,String,String,String,String,String,String):void
addItem(String,String,String,Double):void
addSale(String,String,String,String):void
addProductToSale(String,String,int):void
addGiftCardToSale(String,String,double):void
addServiceToSale(String,String,String,double):void
addSubscriptionToSale(String,String,String,String):void

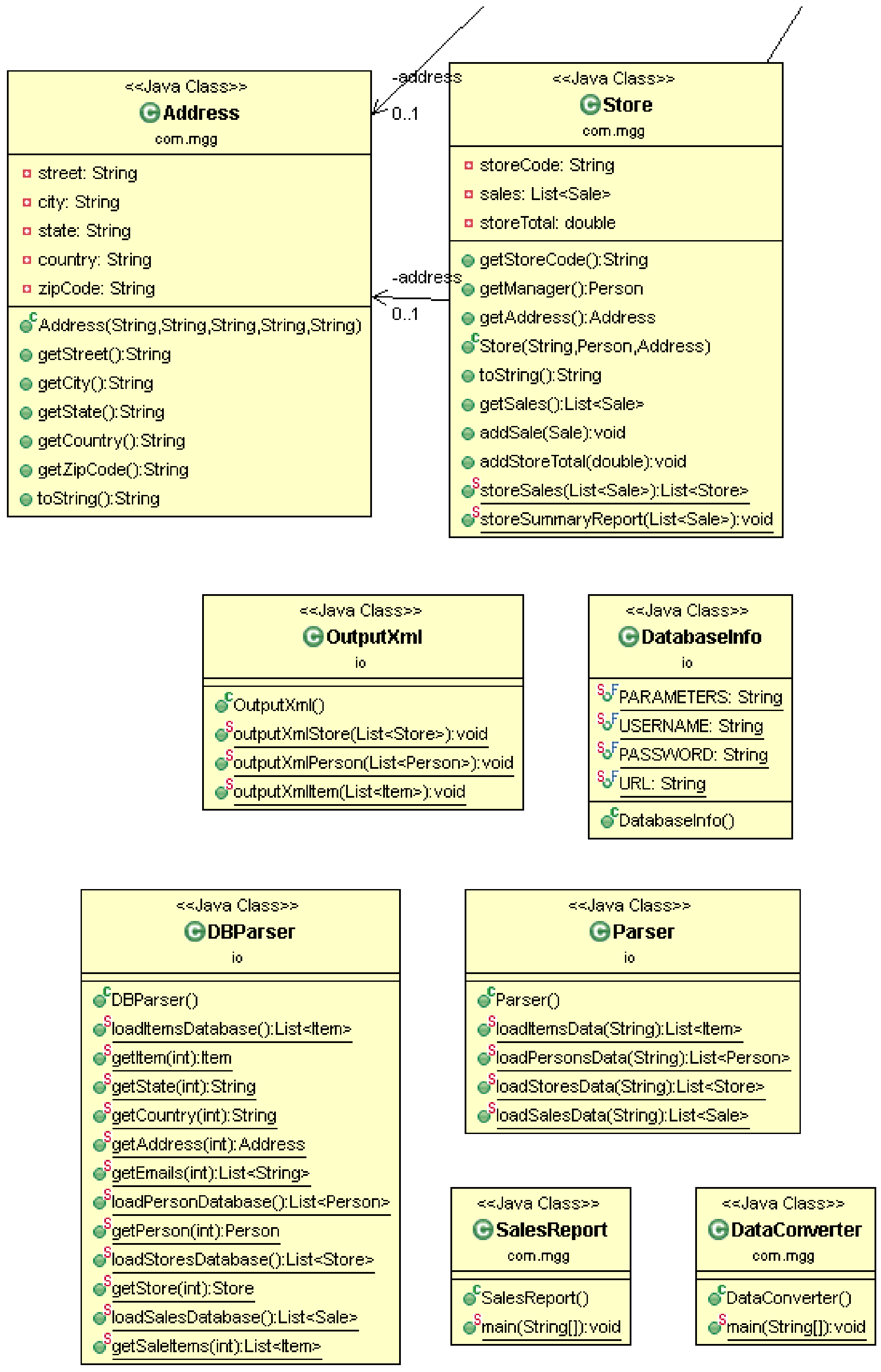


<<Java Class>>
Product
item
Product(String,String)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
NewProduct
item
type: String
basePrice: double
NewProduct(String,String,double)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
UsedProduct
item
type: String
basePrice: double
UsedProduct(String,String,double)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
GiftCard
item
type: String
GiftCard(String,String)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
NewProductSale
item
quantity: int
tax: double
NewProductSale(String,String,double,int)
getQuantity():int
getTax():double
getPreTotal():double
saleInfoToString():String
<<Java Class>>
UsedProductSale
item
quantity: int
tax: double
UsedProductSale(String,String,double,int)
getQuantity():int
getTax():double
getPreTotal():double
saleInfoToString():String
getUsedPrice():double
<<Java Class>>
GiftCardSale
item
preTotal: double
tax: double
GiftCardSale(String,String,double)
getPreTotal():double
getTax():double
saleInfoToString():String



<<Java Class>>
Item
item
code: String
name: String
Item(String,String)
Item(String,String,String)
getCode():String
getName():String
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
Subscription
item
type: String
annualFee: double
Subscription(String,String,double)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
Service
item
type: String
hourlyRate: double
Service(String,String,double)
getType():String
getPrice():double
nameToString():String
saleInfoToString():String
getTax():double
getPreTotal():double
<<Java Class>>
SubscriptionSale
item
tax: double
totalCost: double
beginDate: LocalDate
endDate: LocalDate
SubscriptionSale(String,String,double,LocalDate,LocalDate)
getBeginDate():LocalDate
getEndDate():LocalDate
getTotalCost():double
getTax():double
getDays():long
getPreTotal():double
nameToString():String
saleInfoToString():String
<<Java Class>>
ServiceSale
item
tax: double
numberOfHours: double
ServiceSale(String,String,double,double,Person)
getTax():double
getNumberOfHours():double
getServicer():Person
getPreTotal():double
nameToString():String
saleInfoToString():String


-servicer
0..1
<<Java Class>>
Employee
person
discount: double
salesMade: List<Sale>
Employee(String,String,String,String,Address,List<String>)
getSalesMade():List<Sale>
addSaleMade(Sale):void
getDiscount():double
getType():String
employeeList():List<Employee>
employeeSales(List<Sale>):List<Employee>
employeeSummaryReport(List<Sale>):void
<<Java Class>>
Person
person
personCode: String
lastName: String
firstName: String
emails: List<String>
Person(String,String,String,String,Address,List<String>)
getPersonCode():String
getType():String
getLastName():String
getFirstName():String
getAddress():Address
getEmail():List<String>
toString():String
personNameToString():String
getDiscount():double
-manager
0..1
<<Java Class>>
RegCustomer
person
discount: double
purchase: List<Sale>
getType():String
RegCustomer(String,String,String,String,Address,List<String>)
getDiscount():double
getPurchases():List<Sale>
addPurchase(Sale):void
<<Java Class>>
Customer
person
discount: double
Customer(String,String,String,String,Address,List<String>)
getDiscount():double
getPurchases():List<Sale>
addPurchase(Sale):void
customerPurchases(List<Sale>):void
<<Java Class>>
GoldCustomer
person
discount: double
purchase: List<Sale>
getType():String
GoldCustomer(String,String,String,String,Address,List<String>)
getDiscount():double
getPurchases():List<Sale>
addPurchase(Sale):void
<<Java Class>>
PlatinumCustomer
person
discount: double
purchase: List<Sale>
getType():String
PlatinumCustomer(String,String,String,String,Address,List<String>)
getDiscount():double
getPurchases():List<Sale>
addPurchase(Sale):void



<<Java Class>>
Address
com.mgg
street: String
city: String
state: String
country: String
zipCode: String
Address(String,String,String,String,String)
getStreet():String
getCity():String
getState():String
getCountry():String
getZipCode():String
toString():String
-address
0..1
-address
0..1
<<Java Class>>
Store
com.mgg
storeCode: String
sales: List<Sale>
storeTotal: double
getStoreCode():String
getManager():Person
getAddress():Address
Store(String,Person,Address)
toString():String
getSales():List<Sale>
addSale(Sale):void
addStoreTotal(double):void
storeSales(List<Sale>):List<Store>
storeSummaryReport(List<Sale>):void
<<Java Class>>
OutputXml
io
OutputXml()
outputXmlStore(List<Store>):void
outputXmlPerson(List<Person>):void
outputXmlItem(List<Item>):void
<<Java Class>>
DatabaseInfo
io
PARAMETERS: String
USERNAME: String
PASSWORD: String
URL: String
DatabaseInfo()
<<Java Class>>
DBParser
io
DBParser()
loadItemsDatabase():List<Item>
getItem(int):Item
getState(int):String
getCountry(int):String
getAddress(int):Address
getEmails(int):List<String>
loadPersonDatabase():List<Person>
getPerson(int):Person
loadStoresDatabase():List<Store>
getStore(int):Store
loadSalesDatabase():List<Sale>
getSaleItems(int):List<Item>
<<Java Class>>
Parser
io
Parser()
loadItemsData(String):List<Item>
loadPersonsData(String):List<Person>
loadStoresData(String):List<Store>
loadSalesData(String):List<Sale>
<<Java Class>>
SalesReport
com.mgg
SalesReport()
main(String[]):void
<<Java Class>>
DataConverter
com.mgg
DataConverter()
main(String[]):void

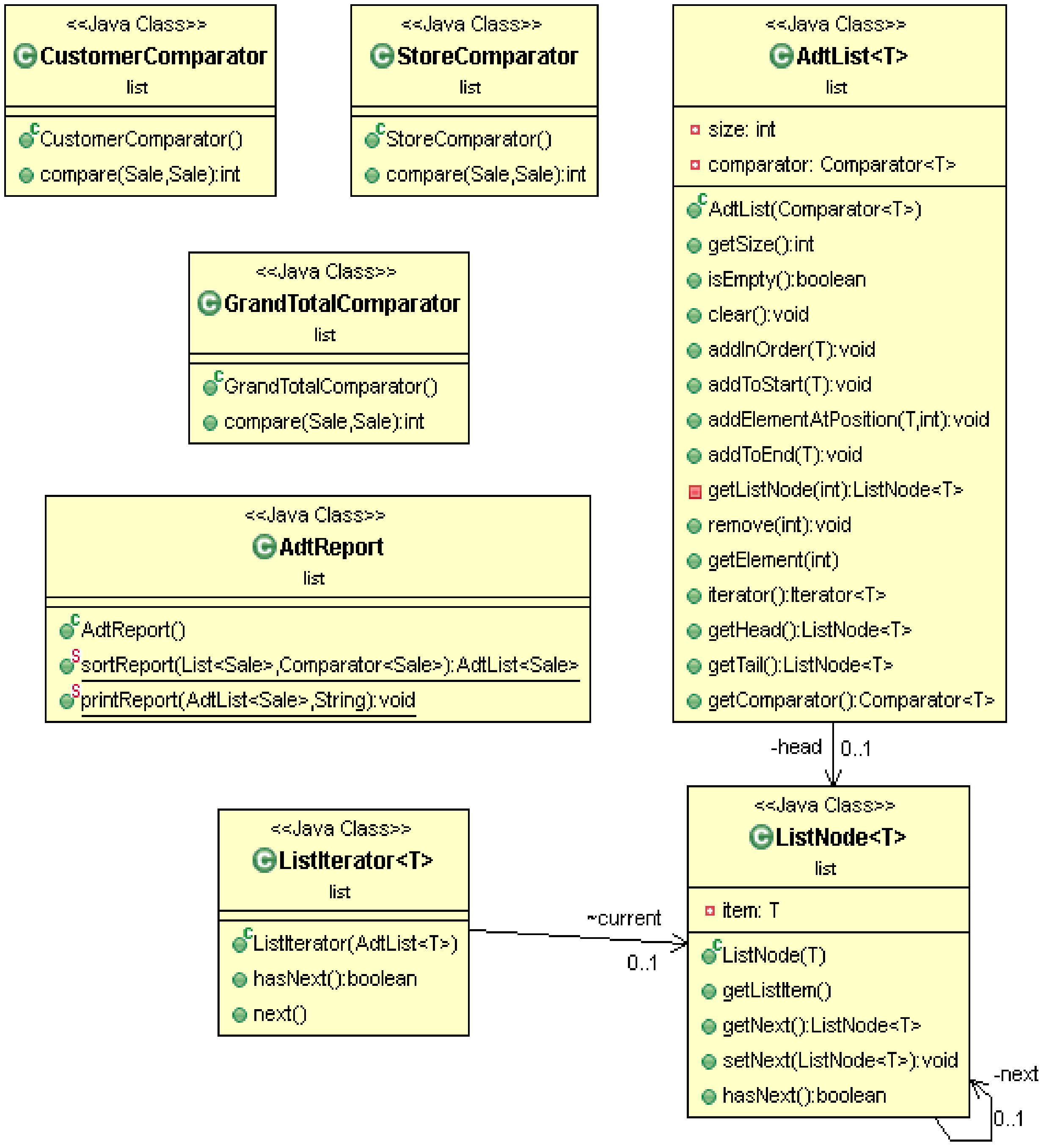


Figure 2: A compilation of the UML Diagram representing the class structure used in our construction of the Sales System for MGG. Image produced using ObjectAid UML generator in Eclipse.

3.2.1 Component Testing Strategy

Testing cases for phase one of the build included custom designed data for the item, person, and store data conversion and output classes. The design for these test cases included generation of csv data based on a given example of data for each class that the previous owner kept. From the csv data, generation of expected XML output helped to compare expected vs actual output in the design process. Various tools were helpful while dealing with the inheritance of the subclasses. During the testing process, the early design did not have adequate inheritance, and thus the tests failed, with incorrect names and titles. After the inheritance was fixed, the next challenge to implement a passing test case was implementing aliasing for some of the types. Most notably the people needed to be aliased because the test cases output a reference to the name, rather than just the name. The test data designed for the database described above, was the same data used to test CSV parsing, and the associated JDBC framework from the database.

3.3 Database Interface

Using JDBC in Java it's possible to make a connection to the database built in MySQL and perform more adaptable data loading than the previous setup would allow for. By making specific methods for getting the state, country, items, saleItems, persons, and stores from the database report outputting can be improved compared to parsing CSV files. In the methods, a driver is loaded, a connection to the database is made, and a query is executed to pull the desired information from the database, set them to useable variables and make objects as needed. These methods also allow for reuse of the reporting functions that were previously designed, since they didn't rely on internal parsing of data. This method design allows for updating and deleting information from the database as well. By setting up the same connection and calling insert statements, it becomes an adaptable way to make any of the objects needed within the database. For objects that are inserted and have foreign key references to other objects, functionality was set up to either get the other object, if it exists already, or make a new object if needed. This is especially useful compared to the old system, where instead of reusable information stored in a database, the previous owner had csv files and macros. With this new system any objects designed in MySQL related to MGG can be inserted and saved in the database.

3.3.1 Component Testing Strategy

Given that we established test cases, when designing the sale reporting functions, the testing for the database interface was fairly straightforward. The expected output should be the same as that designed for reading from the original CSV files. Insert statements used to test in the initial phase of the database development were not needed. The actual output should match expected output with correct java construction of the objects used. Given the expected output that was established in previous phases, the implementation of a database loading class was a success. Some noteworthy items occurred in implementation. The first issue was in including the mysql connectivity jar file in the build path, once this was fixed the driver and connection were established. Another issue that was addressed was in how the system constructed sales, with a Customer type customer, and an Employee type employee. Initially this was included to restrict what type of person could sell and buy. Employees, however, are able to buy items from themselves (depending on the amount). Thus when building sales, it threw an exception when trying to construct an employee as a customer. This led to the change that the customer in sale could be any person type, and the employee stayed an employee. After this implementation no other issues were found.

3.4 Design & Integration of Data Structures

The ADT designed for the MGG system is derived from a linked list. It is a linear list type where each element has a node and the node references the next node in the list. Basic add, remove and retrieval operations are included to facilitate manipulation of the ADT with automatic resizing, so the size need not be fixed. The ADT is parameterized to make it generic and fit any type of object. It also includes a generic comparator in order to maintain a given order when adding or removing rather than sorting it after adding. This is especially useful in that users then don't have to impose a sort each time they add or retrieve information, rather it is given during instantiation of the data structure. The instances of the java.util List type previously used were replaced with the ADT structure for parsing and reporting function. A further look at the reporting type led to the design of a report method for reusability that passed in sort type and comparator type and output the sorted sale list.

3.4.1 Component Testing Strategy

The ADT list was tested using the same test data previously made, but the three reports output sales data sorted by customer, total amount of the sale, and then by store. Initially the testing failed due to not building the comparator correctly and none of the lists were sorted. After fixing the comparators and establishing a reporting method the lists were sorted as expected.

3.5 Changes & Refactoring

Although we have mainly stuck to our origin design, there have been a few times where the design needed to be redone and changed slightly. One of these changes occurred during Phase I of construction. We struggled to get it to function properly and decided to implement the use of inheritance to allow for an easier way to output our data to XML. Originally, we did not use inheritance which proved to be a challenge in getting individual values for different items. In addition, during phase one we used aliasing in Xstream to improve the output. Later it was important to take out the salesperson and manager classes and just have the employee class, since most of our use was with that class. We also changed our design to include an intermediate product class that holds the base price unit shared by the used product, new product and gift card classes.

4. Bibliography

[1] Bourke, Chris. (2020). *Computer Science II.* Department of Computer Science & Engineering, University of Nebraska Lincoln

[2] Cbourke. (2021). *Chris Bourke* [Youtube Channel]. Youtube. Retrieved March 25, 2021, from https://www.youtube.com/user/cbourke3