DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

# The Matrix Sales System

## Computer Science II Project

**[Pratik Pande, Adrian Melgoza]**

**12/03/2021**

**[Final Version]**

This document outlines the process of creating a new database-backed sales system application for The Matrix computer store.

## Revision History

[This table tracks the various major changes to this document]

| Version | Description of Change(s) | Author(s) | Date |
|---|---|---|---|
| 1.0 | Initial draft of design document, including introductory information. | Adrian Melgoza, Pratik Pande | 2021/09/17 |
| 2.0 | Updated document after the creation/implementation of classes to model store attributes. | Pratik Pande, Adrian Melgoza | 2021/10/01 |
| 3.0 | Third draft of the document detailing adjustments made to classes, and the generation of summary reports. | Adrian Melgoza, Pratik Pande | 2021/10/15 |
| 4.0 | Updated document with information regarding the process of designing and implementing an SQL database to create a data model. | Pratik Pande, Adrian Melgoza | 2021/10/29 |
| 5.0 | Updated document regarding an update to the application which allows it to interact with the created database. | Adrian Melgoza, Pratik Pande | 2021/11/12 |
| 6.0 | Final draft of design document, with all required updated information, including a breakdown of the incorporation of an API and ADT. | Pratik Pande, Adrian Melgoza | 2021/12/03 |

# Contents

# 1. Introduction

This document addresses the creative process for a database-backed application, designed for The Matrix computer store. Upon the acquisition of the store chain, new ownership realized that the previous infrastructure to maintain store records was vastly outdated. Therefore, this application is being designed to accommodate for updated business practices that will be implemented in order to allow the store chain to enter the competitive market and renew the framework.

## 1.1 Purpose of this Document

The purpose of this document is to outline the technical design process of the Matrix Computer Store database-backed application in Java. This application has been created from scratch, is object-oriented, and written in Java. Its main purpose is to:

- Improve the manner in which records are maintained by modernizing the system using a database backed sales system
- Support the Matrix Computer Store's business model as accurately as possible

This document will be updated as needed depending on how requirements change over the course of the development of the application. Descriptions of the configuration and installation of various tools such as Oracle and SQL are NOT included in this document.

## 1.2 Scope of the Project

This application successfully implements an updated framework to better manage the infrastructure of The Matrix computer store. Previously, records were maintained using outdated macros and Excel spreadsheets, reducing the efficiency of tracking sales activity at each store, and thus making it nearly impossible to gather data which could be analyzed and improve the business model. Problems also arose in keeping records of this data. This application is being developed for the new owner of the franchise, Thomas Anderson, who required that the new implementation be object-oriented, written in Java, represent all artifacts within the system efficiently, and in doing so, allow the store chain enter the competitive market.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

Abstract Class: A restricted class that cannot be used to create objects (to access it, it must be inherited from another class)

- Can contain both abstract and regular methods

Interface: An abstract type that is used to specify a behavior that classes must implement

Persistence: To store permanently

### 1.3.2 Abbreviations & Acronyms

EDI- Electronic Data Interchange

XML- Extensible Markup Language

JDBC- Java Database Connectivity

API- Application Programming Interface

ADT- Abstract Data Type

DDL- Document Data Language

# 2. Overall Design Description

This application has several important components that constitute its complete functionality. Major aspects of the design include:

- Proper Data Representation- Modeling all attributes of The Matrix with their respective qualities
- Summary Reports- Representation of aggregate data to ensure the relationships defined between objects have been implemented correctly
- Database Design- Designing a relational database to model the previously defined objects
- Database Connectivity- Refactor code to load data to database rather than flat files
- Database Persistence- Implementing and using an API to save data to the database
- Sorted List ADT- Design and implement a sorted list ADT and integrate it into the application

## 2.1 Alternative Design Options

In order to model the traits present in the The Matrix database, the initial plan consisted of simply three superclasses which would hold the required information. However, this idea was improved upon with the generation of multiple subclasses to better hold more specific characteristics of different instances of object types. The chosen method is more advantageous because it allows for more flexibility and specificity. Since multiple types of an object exist, with overlapping qualities, it is better to create a superclass for their overlapping traits and further subclasses to represent more specific characteristics, rather than trying to include such differentiation all in one object representation.

Additionally, the original idea was that the initial structure of the program would remain intact and the objects that were in place would be used to continue working on the store model. However, due to the design of the program from the first phase, certain aspects of the model were too restrictive. Therefore, it was necessary to generate an additional group of subclasses so that more specific aspects of types of customers, and products could be captured and accurately depicted in the generated reports. Changes also had to be made to the file parsing process, and new classes had to be created to generate the reports according to updated requirements.

When generating sales reports, the formatting of the information was not correctly composed, and therefore was also adjusted by changing the procedure in which the information was yielded. This required a simple change in the format of printing information, which in turn, allowed for more control over the presentation of the information which better conformed to the guidelines in place.

After the initial completion of the database, issues arose in connectivity between tables. Initially, the tables had been structured in such a way that each address had to have a person and store attached to it, which was unnecessary, and restricted the capabilities of proper data storage and management. This was adjusted to handle addresses for persons, and addresses for stores independently, while still making use of a single address table structure to maintain efficiency.

# 3. Detailed Component Description

The following chapter details the implementation and characteristics of individual components that constitute the overall application. Each section also includes the approach that was taken to test each part of the design to ensure that its performance meets the specified standards.

## 3.1 Database Design

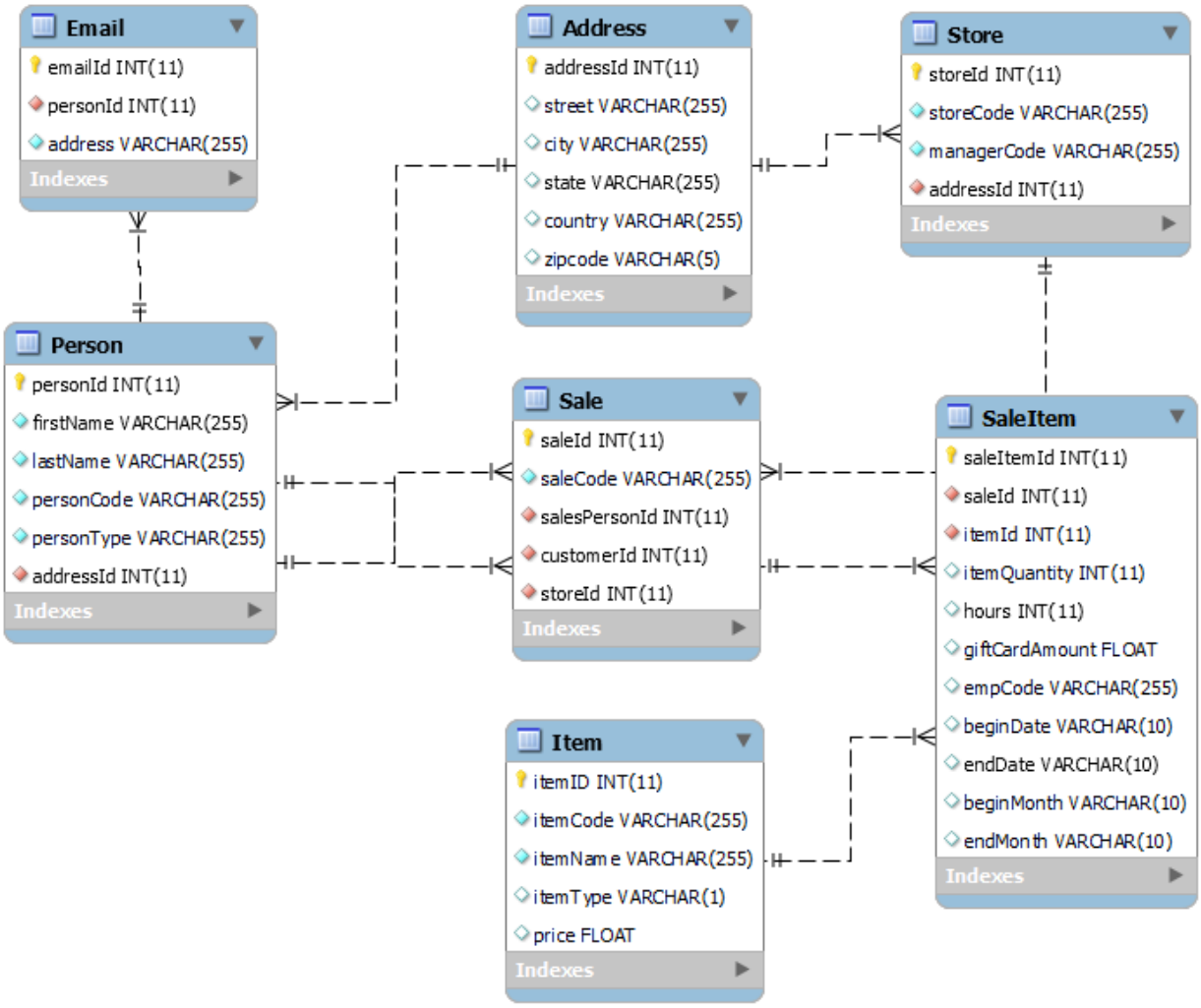


**Figure 1: The figure above is a representation of the structure of the database. Crow's foot notation allows for an easy-to-interpret symbolism of "one -to-many" and "one-to-one" connections.**

Upon the completion of the application framework (see section 3.2), a data model was designed to support the application. This was done by designing and implementing an SQL database using a DDL file. The DDL file contains several queries to both create the framework of the database in the form of tables, and insert data into the tables themselves. It was necessary to create tables that could easily support the application designed in the previous phase, and the following guidelines were used as a basis for design:

- Preserving data integrity through the usage of foreign, primary, and non-primary keys as needed
- Defining the appropriate relationships between tables to replicate the interactions that occur during a given sales transaction while simultaneously managing all relevant unique cases

In order to create the most efficient database for the Matrix, while keeping in mind the franchise's business practices, a more simple database composition was deemed appropriate (Figure 1). By creating only seven tables in this database, it was made more efficient and easily usable, while still accurately preserving all essential information. The seven tables used in the database represented the following units: Emails, Addresses, Persons, Stores, Sales, Items, and SaleItems, which for the most part, had nearly identical attributes in comparison to their

object counterparts in the application. For example, the email table contains an email ID, person ID, and an address for the email itself. By using this design it allows for multiple emails to be associated with a single person, which in turn makes it easier to sort and gather information from customers and employees. Tables are also linked to one another efficiently by taking advantage of keys. Foreign keys are present in the necessary tables which refer to a primary key in another table, and thus allow relevant information to be shared between tables. For example, each person whose information is present on the database has an address value that is stored on the same database separately, but is assigned to them with a key. Figure 1 showcases a complete breakdown of the database structure, detailing the fields that are present in each table, as well as the connections that are established between relevant tables.

### 3.1.1 Component Testing Strategy

In order to test the configuration of the database, a separate artifact outlining several more queries was created. Each query retrieved a different piece of information from the database, and therefore served as a method of testing the structure and integrity of the model once it was implemented. This helped in determining both if information had been stored properly, and confirmed that it could be retrieved without fault.. By testing the validity of the database through extracting specific information, it was possible to determine where faults occurred, if any, in the schema, tables, and ensure the responsiveness of all entities involved.

## 3.2 Class/Entity Model

In order to model real-world entities and situations as efficiently as possible, classes have been created to replicate all components of a given relation. Each object stores information pertaining to it. For example a Person has a unique alphanumeric code, a second code to designate whether they are a type of customer or an employee, their first and last name, email address(s), and a residential address. Due to the fact that there are multiple types of people that may need to be classified, the Person class was made into an abstract superclass, to allow any overlapping information between subtypes to be accommodated. Since every type of person has the attributes mentioned above, they are all included in the singular Person entity. Any unique attributes they may have, such as a list of all recorded sales for a given employee, are stored in their own class. This allows for reusability in the program, and therefore each class builds upon an existing class. The aforementioned approach was generalized and applied to all objects that required such a structure.

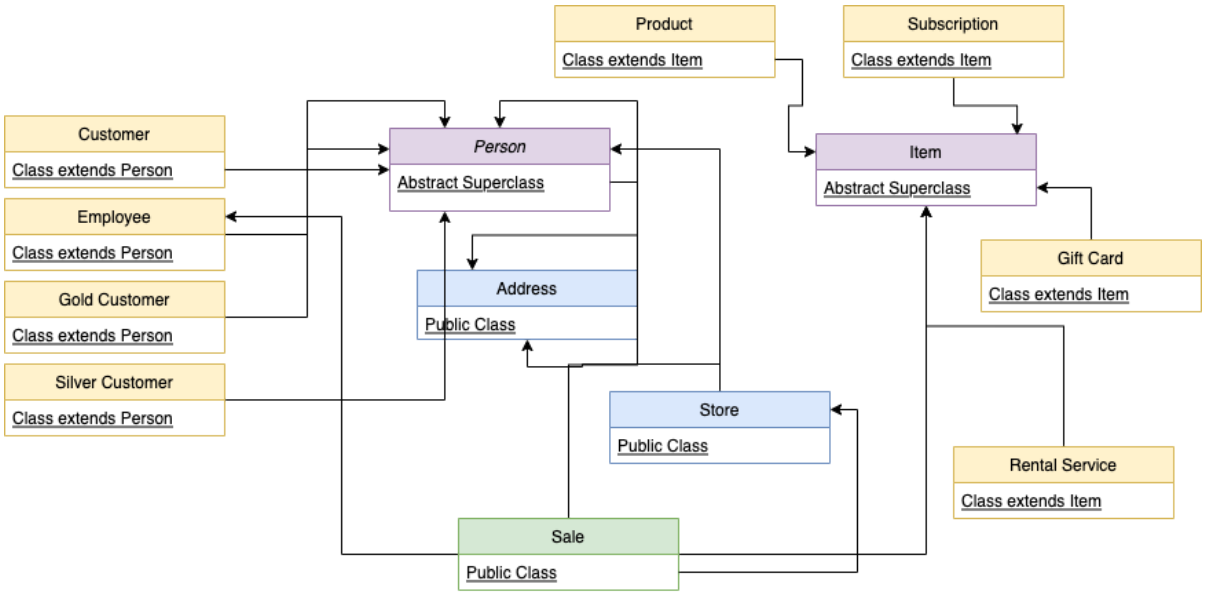


Figure 2: This diagram showcases the relationships among the classes in the application that model real-world entities and information.

Present in each class are methods that accomplish several tasks, each relating to the role of the class in the overall formation of the application. Getters have been generated in order to access certain data in a way that protects it

and prevents changes to data from damaging other elements in the application elsewhere. A constructor is also present in each class, which allows the objects and their attributes to be initialized.

### 3.2.1 Component Testing Strategy

The testing of objects in the The Matrix application was completed through various means. Since each entity represented a given real-life element in The Matrix computer store, several test cases were developed over the course of multiple phases in order to make sure that each class had proper encapsulation and attributes. Methods to return specific attributes of an object were repeatedly tested. Additionally, thorough evaluations were performed to make sure classes could appropriately interact with one another. Mock data was generated to imitate real sales interactions, and was used to establish whether or not objects had been correctly and efficiently implemented. This was done by feeding the generated data to the program and validating the results multiple times, making sure that each time the guidelines and standards for that phase were met, and modifications were made accordingly. For information regarding class/entity design choice, refer to section 2.1, and for a further breakdown of each object, refer to section 3.2.2.

### 3.2.2 Class/Object Structure Breakdown

This section provides an explicit description of each class used in the application to represent a real-world entity. Certain objects have been grouped together based on their standing in order to preserve the organization and coherence of the document.

Person:

- An abstract class containing all general information about customers and employees
    - This class is abstract to allow for multiple subclasses to extend to it
        - Essentially, this preserves broad attributes while simultaneously providing lower-level classes with the freedom to implement abstract methods suited to their more specific needs
    - The Address variable referenced in this class is an entity of its own, due to having multiple elements itself (rather than storing each element individually in the classes that make use of it)

Employee/Customer/Silver Customer/Gold Customer:

- These subclasses extend the Person superclass and therefore inherit the main attributes and any abstract methods that may be present in the Person class
- Each customer has a unique discount depending on their standing, hence the separation into their own distinct subclasses
    - A normal customer has no discount
    - Silver customers receive a 3% discount, whereas Gold Customers receive a 7% discount
- Each employee shares the same main attributes as customers however there are additional characteristics described in the class
    - Each employee receives a 10% discount
    - A list of sales conducted by a given employee is also present
        - This takes advantage of the defined “Sale” object (full description is present later in this section)
    - A running sum of total sales is also present

Item:

- An abstract class containing all general information about items that can be purchased
    - This class is an abstract class so that multiple classes can extend to it
    - Includes an item code, type, and name

- Includes abstract methods to acquire the price, quantity, and tax amount of a sales amount, which allows these methods to be more specifically implemented for respective types of items

Product/Rental Service/Service/Subscription/Gift Card:

- Each of these classes inherit the main methods and variables of the Item abstract, and therefore have the ability to take advantage of implementing methods to specifically cater to their attributes
    - Each product has a differing payment method, therefore each subclass' method implementation can differ without affecting the abstract class
- Each class has its own attributes in addition to the characteristics it inherits therefore preserving the fact that it is a model of a more specific type of item in the real-world

Sale:

- A class storing the attributes of a given sale
    - Code (each sale is assigned a code, therefore all the relevant details of a sale can be accessed code is used to search for the matching sale)
    - Store (store where the sale occurred)
    - Customer and Employee involved in the transaction
        - Attributes of both of these object types are preserved and relevant information is stored for each object (ex. The sale is added to the list of sales present in the Employee class for the particular employee involved)
    - List of items present in the transaction
        - Since it is a list of type "Item", the attributes of the Item object are preserved

Store:

- A class storing the attributes of a given store
    - Code
    - Manager (type of Employee)
    - Address (makes use of the address object rather than storing each individual element of an address separately)
    - A list of type "Sale"
        - Each sale at a given store location is added to this list
        - Due to Sale being a predefined object, no other aspects of a sale have to be stored separately, as all of the attributes are already managed in the Sale class

## 3.3 Database Interface

A new design condition required a modification of the application to allow reading from the previously designed database rather than from a flat file. The database had already been tested thoroughly in preparation for establishing this connection, therefore it was now in a state where the application could be configured in order to do so. In order to modify the application to connect to the database, the JDBC API was used. The driver class was changed to connect to the database, load the appropriate data, and create the appropriate objects, rather than reading the data from flat files (Figure 3). The aspects of the class that were used to generate reports and summaries retained their functionality.

```java
public static String getPersonCodeFromId(int personId) {
    String code = null;
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(DatabaseConnection.URL, DatabaseConnection.USERNAME,
            DatabaseConnection.PASSWORD);
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    // 3. create a query
    String query = "Select personCode from Person where personId = ?;";
    PreparedStatement prep = null;
    ResultSet rs = null;
    try {
        prep = conn.prepareStatement(query);
        prep.setInt(1, personId);
        // execute
        rs = prep.executeQuery();
        // process results
        while (rs.next()) {
            code = rs.getString("personCode");
        }
        rs.close();
        prep.close();
        conn.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return code;
}
```

**Figure 3: A depiction of a modified method which now reads from the database and executes a query to retrieve the desired information. In this case, a query to get the unique code from a person who has a given assigned ID in the database.**

Adjustments to the program as seen in Figure 3, were made to access all relevant information that was previously extracted from a flat file. Essentially, the manner in which data was being stored had evolved, and thus it was necessary to alter the process by which that data was being interpreted and retrieved.

### 3.3.1 Component Testing Strategy

For testing purposes, data was added, modified, and removed from the database (when it came time for the final implementation of the application, the data was restored to the values needed to generate reports). The database was populated with a set of known data so that the exact outcome of each query was ensured. Thus, it was possible to relate the outcome with the predetermined result, and if the condition was not satisfied, changes were made. Further testing was conducted to measure the efficiency of queries and to establish that the correct set of data was being targeted. Additionally, it was ensured that the data that was returned successfully contained all of the required information and was preserved in the correct form. While constructing this connection it was ensured that all aspects of the program were well documented and integrated.

## 3.4 Design and Integration of Data Structures

After the implementation of the database interface, additional modifications were made to handle persistence and deletionI. A new class was created to handle these interactions, and was configured with various utility methods in order to define a general API which could use the JDBC API to link with the database and perform their designated tasks. Methods were defined to include the following functionality:

- Remove or add some/all records of a particular kind to/from the database
- Remove records associated with a given code
- Retrieve a particular record associated with a given code
- Clear all tables in the database
- Sort sales in alphabetical order of customers last names

- Sort sales in order of largest total cost

### 3.4.1 Component Testing Strategy

For testing the addition of a linked list for sales, new sales were made to be added to the new SaleList. After inserting the new data, it was printed out in order to make sure that the list was in the correct order with both alphabetical order of the customer's last name and the total price of the sale itself. The data was predetermined, so if the list was not in the correct order it would be able to be checked easily and fixed according to the problem. Testing was done two times for the linked list, the first was testing the alphabetical order of the customers. When doing this there were only two errors that occurred. The first error was in the function that added the sale in the correct order; in this function it was passing through a null sale. To fix this a simple error statement was added to the function that made sure no null statement would be able to pass through again. The second error was the actual order of the sales itself. The list was being ordered in the completely wrong way, it was ordered with the latest letter being added first and the earliest letter being added last. In order to fix this, the solution was just simply flipping the order of the less than or greater than signs in the function that was added according to the alphabet. None of these errors were enough to force a redesign. The first error affected the development time as it took a while to understand how to fix my error statement to not pass through a null value. There were no other errors in the implementation of the linked list for sales.

## 3.5 Changes & Refactoring

During the development of this system, there was only one design that needed to be dramatically changed. The database design was completely changed in order to accommodate the data. The first design of the database was poor and had many unnecessary tables like employee, customer, and storesale. When the system needed to be able to add information from java a significant change was needed to make the process of addition and deletion of information more efficient and convenient. This change was the only drastic change needed in order to maintain efficiency and convenience.

# 4. Bibliography

This section provides a bibliography of all materials referenced during the creation of this application.

[1] *Bourke, Chris. Computer Science II. 3rd ed, Chris Bourke, 2021.*

[2] *Bourke, Chris. "Databases and SQL." Computer Science II- Introduction to Databases. https://cse.unl.edu/~cbourke/ComputerScienceII/modules/module-SQL/. Accessed 2021.*

[3] *"Relational Database Design." Wikibooks, Open Books for an Open World, Wikibooks, 19 Oct. 2021, https://en.wikibooks.org/wiki/Relational_Database_Design.*

[4] *"SQL Tutorial - Learn SQL." SQL Tutorial - Learn SQL, Art Branch Inc., 2015, http://www.sql-tutorial.net/.*