## Overview

We have implemented a set of classes to create software for an investment firm, which offers clients the opportunity to buy or sell stocks. The firm assigns one advisor to every client who understands the client's requirements and then assigns a profile to maximize the client's profit. The software provides numerous functionalities, such as adding clients and advisors and creating accounts, and profiles. Additionally, it includes various methods for generating reports and conducting analyses on individual accounts, as well as across all accounts. We use the MySQL database management system to store the data. Please refer to the assignment specification for details on the required methods.

The problem decomposes into three main components: the basic method, which creates clients, advisors, accounts, and facilitates trading stocks; the next component handles reporting, such as the profile of each account, account and advisor value, and more; and lastly, there's the analysis part, which describes stock recommendations to clients and groups advisors with similar interests.

This software handles two types of trades: full and partial purchases of stock. This means that if a client does not have enough money to buy an entire stock, the firm will purchase the full stock from the respective company and then give the client a fraction of that stock.

## Files and External Data

The implementation is divided into classes represented below:

InvestmentFirm: It serves as a central hub for managing investment-related operations within a firm. It provides functionalities for defining sectors, stocks, and investment profiles, as well as adding clients and financial advisors. Additionally, it enables trading shares, tracking account values, providing recommendations, and grouping advisors based on specified criteria.

MysqlConnection: This class is designed to handle connections to a MySQL database, implementing the `DBConnection` interface. It follows the singleton pattern to ensure there's only one instance of the database connection throughout the application. The class provides methods to start, close, and execute queries on the database. It also uses logWriter to write log on .txt file located at the desktop.

TxtLogWriter: This class provides a singleton instance for writing log messages to a text file located on the user's desktop. It ensures thread safety during object creation using synchronized blocks and handles logging of messages with timestamps, types (such as Success, Failure, Information, Warning), and corresponding modules.

UserImpl: This class implements the `User` interface and provides functionality to add a new user to the database. It also performs user validation.

SectorImpl: It implements the Sector interface and provides functionality to define a new sector. It verifies if the sector already exists in the database and if not, it creates a new sector entry with a unique name. It handles exceptions such as SQL errors and custom exceptions related to sector creation.

ProfileImpl: It implements the Profile interface and is responsible for adding a new profile to the system. It validates the sectors associated with the profile, ensuring they exist in the database, and then inserts the profile details along with its sector holdings into the database.

AccountImpl: It implements the Account interface and provides functionality for managing accounts within the system. It includes methods to create a new account and change the advisor associated with an existing account. These methods handle validation of account details such as investor and advisor IDs, account names, and profile names.

StockImpl: It implements the Stock interface and handles operations related to stocks in the system. It includes methods for adding new stocks to the database, setting stock prices, trading shares, and disbursing dividends. These methods handle input validation and database transactions, to ensure the reliability of stock-related operations within the system.

Dividend: It facilitates the process of disbursing dividends to shareholders. It retrieves account details of shareholders eligible for dividends based on the provided stock symbol, calculates the number of shares the firm needs to buy to manage dividends, and updates shareholder accounts with dividends either as cash or reinvested in additional stocks. Additionally, it updates the stock history to reflect dividend disbursement.

Trade: It manages stock trading operations, including validating stock symbols and account information, buying and selling stocks, and updating cash balances. It ensures the validity of stock and account information before executing transactions, handles both buying and selling operations, and updates account details such as cash balance and stock holdings accordingly. Additionally, it provides functionality to update cash balances for cash-in and cash-out transactions.

ReportingImpl: It implements reporting functionalities for financial accounts, including calculating account values, advisor portfolio values, investor profits, and sector weights. It also identifies divergent accounts based on assigned and actual sector profiles. The class ensures data validity by checking the existence of users and their roles before performing reporting operations.

DivergentAccounts: It is responsible for fetching data related to assigned and actual profiles for divergent accounts. It retrieves assigned profiles from the database, including sector names and their percentages, and actual account profiles, including cash, stock holdings, and sector information. These data are used to identify divergent accounts based on variations between assigned and actual profiles

AnalysisImpl: It provides functionalities for stock recommendations based on cosine similarity analysis. It calculates recommendations for a given account by comparing its stock holdings with other accounts, considering factors such as stock popularity and similarity.

CosineSimilarity: It is responsible for calculating the cosine similarity between different accounts based on their stock holdings. The calculations involve iterating through the stock holdings of each account and applying the cosine similarity formula to determine the similarity value.

Recommendation: It is responsible for generating stock recommendations for a given account based on various factors such as stock popularity, current stock holdings of the account, and the number of comparisons considered. It implements methods to prepare data for recommendation, refactor account stock holdings to ensure consistent dimensions for comparison, calculate stock popularity, recommend stocks based on popularity and current holdings, and convert stock IDs to stock symbols for presentation. These methods together provide a comprehensive recommendation system for stock trading.

CustomException: It extends the built-in Exception class. It provides a constructor that allows you to set a custom error message when throwing this exception.

Dao Classes *(AccountDao, DivergentAccountDao, DividendDao, PopularityDao, ProfileDao, RecommendationDao, SectorDao, StockDao, TradeDao, UserDao)*

- Encapsulate data access logic and provide an interface for interacting with specific data entities in the system

## Data structures and their relations to each other

AccountDao: Account information with attributes such as account ID, investor ID, advisor ID, profile ID, profile name, account name, reinvestment status, and cash balance.

DivergentAccountDao: Divergent account information, including profile ID, cash balance, and a map of sectors with corresponding values like number of stokes * Price or directly store percentage.

DividendDao: Dividend-related information, including account ID, cash balance, reinvestment status, number of stocks that the account holds,  ACB, stock ID, and stock price.

PopularityDao: Information related to stocks, including stock ID, popularity score (indicate how many accounts hold that particular stock), and a flag indicating whether it's for buying or selling (true - Buy, false - Sell).

ProfileDao: Managing profiles, including profile ID, profile name, and a map of sector holdings where each sector is mapped to its corresponding integer value representing the holdings in that sector.

RecommendationDao: Including available cash in that account and a map of stock IDs mapped to the count of stocks they hold.

SectorDao: Managing sectors, including sector ID and sector name.

StockDao: Information, including stock ID, stock symbol, company name, sector name, sector ID, and price. It also contains enums for stock history type and stock transaction type to indicate different types of transactions and events related to stocks.

TradeDao: Information, including account ID, stock ID, shares exchanged, cash balance, current shares, price, trade type, and ACB.

UserDao: Information, including user ID, name, and role. It contains an enum for roles (either INVESTOR or ADVISOR).

## Assumptions

1) Names are unique for both clients and advisors (case insensitive).
2) The database is not created automatically once the system starts; first, we have to ensure our database exists in MySQL Workbench.

## Choices

1) I used Data Access Object to fetch and transfer data from one method to another, reducing the size of the method parameters.
2) I used to open and close MySQL connections in each method to free up unused resources, allowing the server to handle other systems in a multithreading environment.
3) I use transactions in cases where I need to update more than one table within one method to maintain the consistency and reliability of the data.
4) I used custom exceptions to handle errors efficiently and log the messages to identify where the errors occurred.
5) I also wrote logs in a .txt file located on the desktop, documenting operations with MySQL. This helps track transactions performed by any class, and in case of failure, we have all the necessary information.
6) I use the XAMPP server to access data locally.
7) I store all string-type data in uppercase only.

## Key algorithms and design elements

<u>Cosine Similarity</u>
Step 1: First, create two vectors with the same dimensions, representing the count of stocks with their respective stock IDs for both accounts.
Step 2: Then, perform a dot product of the stocks present in both accounts.
Step 3: Additionally, calculate the square of the count of stocks separately for each account.
Step 4: Next, take the square root of the values obtained from step 3 for both accounts.
Step 5: Now, check if any of the numbers from Step 4 is zero. If yes, directly return 0; otherwise, proceed to the next step.
Step 6: Calculate the cosine similarity by dividing the value obtained from Step 2 by the value obtained from Step 4.

<u>Recommendation</u>
1. Create arrays for stockID, popularity, and buy or sell from the stokesPopularity data.
2. Initialize a map to store recommended stocks with their buy/sell status.

```
3. Determine halfComparators as numComparators divided by 2.
4. Iterate through each stock in the popularity data.
   a. If maxRecommendations is 0, exit the loop.
   b. Iterate until encountering a different popularity.
   c. If no common popularity,
         Step 1: If there is a case of a buy then check popularity must be greater than halfComparators and the account does not hold that stock.
         Step 2: If there is a case of Sell then check popularity must be greater than halfComparators and the account must hold that stock.
   d. If common popularity exists:
      i. If the number of common stocks exceeds maxRecommendations, skip.
      ii. Otherwise, iterate through each common stock.
          - Check the same condition as we see in step 4 of section (c)
5. Return the map of recommended stocks.
```

## Limitations

1) In our system, we only buy stocks, but we don't have information about the sellers. In reality, we can only buy stocks if someone is selling them. However, in our system, we can buy as much as we want as long as the account has sufficient funds.
2) We cannot fix any specific amount for a stock. In reality, every company has a fixed number of stocks, and people trade them by buying and selling.
3) Currently, users have no option to reset their reinvestment value. Once the account is created, the set value remains final for their lifetime. However, we need to provide clients with the choice to turn this reinvestment option on or off at any time.
4) Similarly, the advisor doesn't update the user's profile according to changes in market value.
5) It only recommends stocks based on comparisons with other accounts. Instead, we should recommend stocks based on their historical performance and predictions about the future of the sector to which those stocks belong.
6) In the real stock market, there is a brokerage charge per transaction, but here in this software, there is no concept of transaction fees or commissions.

## Updated Database Diagram

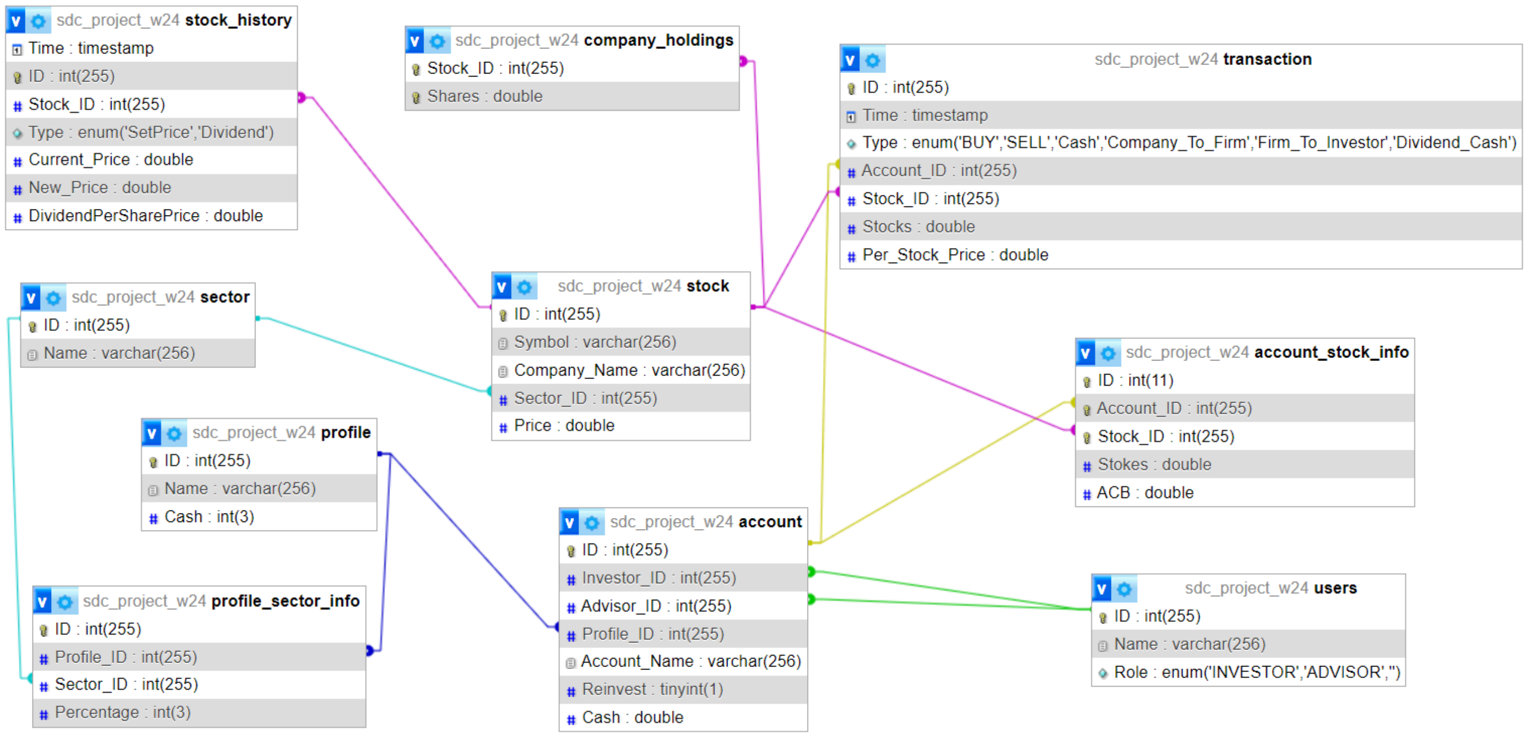

*Figure 1 Database Design*

## References

[1] J. Davis, “JDBC Best Practices”, DZone. [Online]. Available: https://dzone.com/refcardz/jdbc-best-practices. [Accessed: April 15, 2024].