CSCI 3901 Final Project

Name: Rushil Borad

Banner ID: B00977837

Contents

Overview

Investment Firm Management System:

This Java-based system empowers investment firms to streamline client portfolio management and gain valuable insights. Here's what it offers:

- **Client Management:** Effortlessly register clients, assign advisors, and store personalized investment profiles.
- **Investment Tracking:** Monitor client holdings across sectors, calculate sector proportions, and manage dividend reinvestments with fractional share tracking.
- **Portfolio Analysis:** Identify advisors with similar strategies, pinpoint client portfolios that stray from their profiles, and generate data-driven stock recommendations tailored to each client's investment landscape.

This system goes beyond basic portfolio management. By leveraging data analysis, it empowers firms to:

- **Boost Efficiency:** Automate tasks and gain real-time insights for better decision-making.
- **Optimize Strategies:** Identify top-performing advisors and their investment approaches.
- **Reduce Risk:** Proactively address potential portfolio deviations for improved client outcomes.
- **Personalize Recommendations:** Give stock suggestions based on client profiles and similar investor holdings.

The Investment Firm Management System equips firms with the tools they need to manage client portfolios effectively, optimize investment strategies, and ultimately, achieve better financial outcomes for their clients.

Files and external data

Here's a overview of the classes and their functionalities:

1. **DatabaseAccess Package:**
    - Database Connector: Establishes and manages connections to the database.
2. **Databasefunctions Package:**
    - DatabaseHelper: Provides utility functions and methods to interact with the database.
3. **DataEntry Package:**
    - **Sector**: Handles the definition and management of investment sectors.
    - **Stock**: Manages the definition and pricing of stocks traded on the stock exchange.
    - **InvestmentProfile**: Defines and updates investment profiles based on sector allocations.
    - **Advisor**: Declares a financial advisor working for the firm and returns an identifier for the advisor for future reference.
    - **Client**: Declares a client of the investment firm and returns an identifier for the client for future reference.
    - **AccountManager**: Opens a new investment account for a client, managed by a financial advisor. Associates the account with a specific investment profile and reinvestment option. Also reassigns the management of an investment account to a new financial advisor.
    - **TransactionManager**: Opens a new investment account for a client, managed by a financial advisor. Associates the account with a specific investment profile and reinvestment option.
4. **SystemReporting Package:**
    - **ManageProfits**: Computes and reports profit/loss for clients' investment accounts.
    - **PortfolioManager**: Calculates and reports account and portfolio values for clients and advisors.
    - **ReportAccounts**: Generates various reports, including account statements and performance summaries.
    - **SectorWeights**: Computes the proportion of account value held in each sector.
5. **SystemAnalysis Package:**
    - **Recommendations:** Contains algorithms and methods to generate investment recommendations.

- **Groups:** Implements algorithms to identify clusters or groups of similar entities within the investment system.

6. **Main Class: InvestmentFirm**
    - **defineSector(sectorName: String): boolean**: Adds a new investment sector to the system.
    - **defineStock(companyName: String, stockSymbol: String, sector: String): boolean**: Defines a new stock traded on the stock exchange.
    - **setStockPrice(stockSymbol: String, perSharePrice: double): boolean**: Updates the price of a stock based on recent trades.
    - **defineProfile(profileName: String, sectorHoldings: Map<String, Integer>): boolean**: Defines an investment profile with specified sector allocations.
    - **addAdvisor(advisorName: String): int**: Registers a new financial advisor with the firm.
    - **addClient(clientName: String): int**: Registers a new client with the firm.
    - **createAccount(clientId: int, financialAdvisor: int, accountName: String, profileType: String, reinvest: boolean): int**: Creates a new investment account for a client.
    - **tradeShares(account: int, stockSymbol: String, sharesExchanged: int): boolean**: Executes a trade of shares for a specified account.
    - **changeAdvisor(accountId: int, newAdvisorId: int)**: Changes the financial advisor associated with a client's account.
    - **accountValue(accountId: int): double**: Retrieves the current value of a client's investment account.
    - **advisorPortfolioValue(advisorId: int): double**: Computes the total value of portfolios managed by a financial advisor.
    - **investorProfit(clientId: int): Map<Integer, Double>**: Calculates the profit/loss for a client's investment account.
    - **profileSectorWeights(accountId: int): Map<String, Integer>**: Determines the sector allocations for a client's investment profile.
    - **divergentAccounts(tolerance: int): Set<Integer>**: Identifies accounts deviating from their target investment profiles.
    - **disburseDividend(stockSymbol: String, dividendPerShare: double): int**: Disburses dividends to accounts and manages share purchases.
    - **stockRecommendations(accountId: int, maxRecommendations: int, numComparators: int): Map<String, Boolean>**: Recommends stocks to buy or sell for a client's account.
    - **advisorGroups(tolerance: double, maxGroups: int): Set<Set<Integer>>**: Identifies groups of advisors with similar investment preferences.

This breakdown provides a clear overview of the classes and their respective functionalities within the investment management system.

Data structures and their relations to each other

Data Structures within the Investment Firm Management System

Since we're utilizing a relational SQL database for data storage, the primary data structures within the system itself reside in the class implementations. These structures act as representations of the data retrieved from the database tables. Here is a breakdown of potential data structures used within the classes:

Portfolio Management and Analysis:

- **AccountHoldings (HashMap<String, Double>):** This structure likely resides within the Account class or a separate PortfolioManager class. It maps a stockSymbol (string) to the corresponding total investment value (double) for that stock within the account.
- **PortfolioSectorWeights (HashMap<String, Double>):** This structure might exist within the Account class or a separate PortfolioManager class. It maps a sectorName (string) to the corresponding percentage weight (double) that sector holds within the account's total value.

Relationships Between Data Structures:

These in-memory data structures mirror the relationships established within the relational database tables.

- **Client and Account:** A Client object can have a one-to-many relationship with Account objects, achieved by storing the clientID within the Account class.
- **Account and Investment Profile:** An Account object is associated with a single InvestmentProfile object, linked by storing the profileName within the Account class.
- **Account and Holdings:** An Account object can hold various stocks represented by the AccountHoldings map, where the key is the stockSymbol and the value is the total investment value.
- **Holdings and Sectors:** The Stock class likely has a reference to the corresponding sectorName for analysis purposes.

Additional Considerations:

- The system also utilizes other data structures like ArrayLists, Maps, TreeMaps etc (for temporary data storage during calculations) or custom classes representing advisor data if not directly stored in the database.

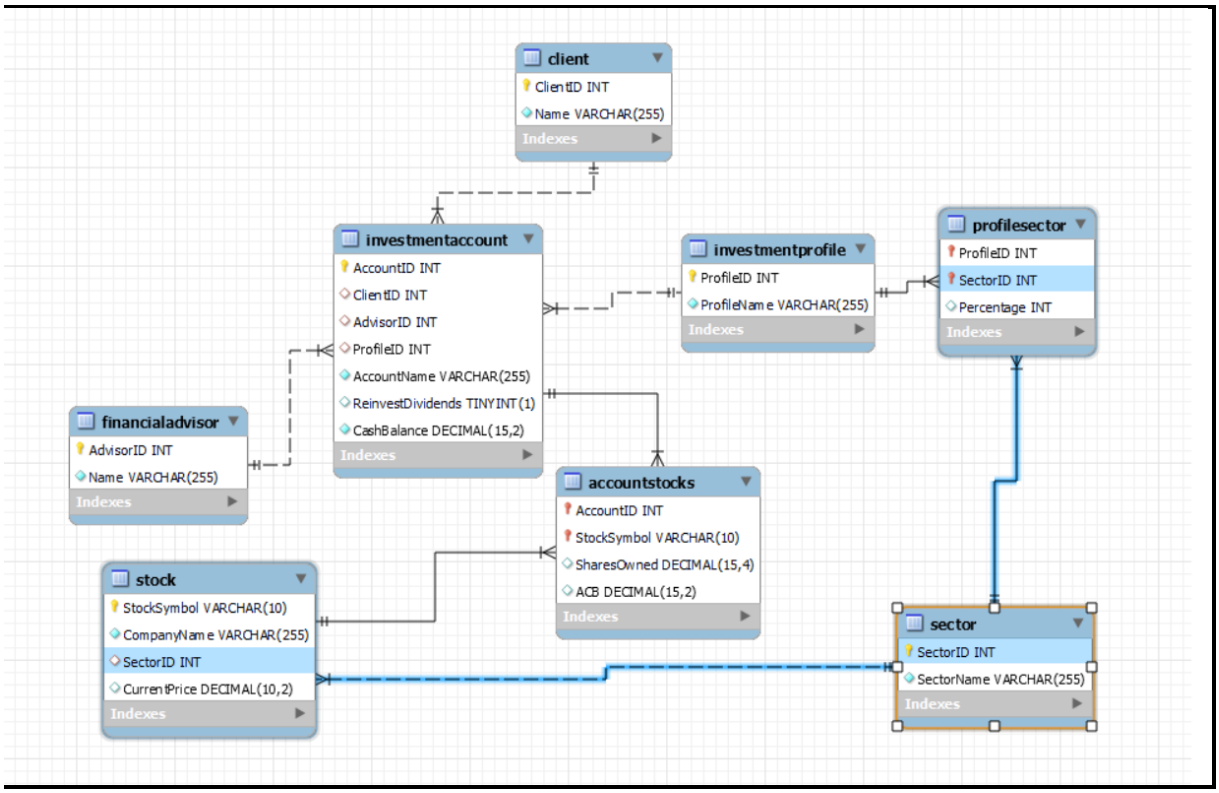
By leveraging these data structures within your classes, you can effectively manipulate and analyze client information, investment holdings, and portfolio data retrieved from the underlying SQL database. Remember to adapt these examples to match the specific attributes and relationships defined in your database schema.

Assumptions

- **Unique Clients:** Client IDs are unique within the system, allowing for efficient client identification and account management.
- **Account Naming:** A combination of a client ID and account name uniquely identifies an account within a client's portfolio.
- **Stock Symbols:** Stock symbols are unique identifiers for stocks within the system, ensuring accurate trade execution and portfolio tracking.
- **Sector Definitions:** A pre-defined set of sectors of cash exists within the system, used for buying and selling new stocks in investment profiles.
- Sufficient Cash Balance Checks: Trade execution verifies that the account has sufficient cash balance to cover the entire purchase, including potential fractional shares.

Key algorithms and design elements

Database ER Diagram:



Here's a breakdown of key algorithms and design elements used in Investment Firm Management System, along with some high-level pseudocode description:

1. Account Creation and Management:

- createAccount(clientId, advisorId, accountName, profileName, reinvestOption)**:**
    - This function performs the following steps:

        - Validates client ID, advisor ID (if provided), profile name, and reinvest option.
        - Checks for a unique combination of client ID and account name.
        - Creates a new account entry in the database with the provided details.

2. Trade Execution:

- tradeShares(accountId, stockSymbol, sharesExchanged)**:**
    - This function involve steps:
        - Verifies the existence of the stock symbol in the system.
        - Retrieves current stock price.
        - Checks for sufficient cash balance in the account to cover the trade (including potential fractional share purchase).
        - Calculates the total cost of the trade (considering whole shares and fractional share).
        - Updates the account's cash balance and stock holdings in the database.

3. Dividend Disbursement:

- disburseDividend(stockSymbol, dividendPerShare)**:**
    - This function follows these steps:
        - Retrieves a list of accounts holding the specified stock symbol.
        - Iterates through each account:
            - Calculates the total dividend amount based on the number of shares held.
            - Updates the account's cash balance based on the dividend amount (considering reinvestment option for fractional shares).
            - If reinvestment is required, calculates the number of fractional shares purchasable with the remaining dividend amount.
            - Updates the account's stock holdings with the purchased fractional shares (if applicable).
        - Returns the total number of accounts that received a dividend payout (including those with no fractional share purchase required).

4. Portfolio Analysis - Sector Weights:

- profileSectorWeights(accountId)**:**
    - This function uses the following logic:
        - Retrieves the investment profile associated with the account.
        - Obtains the sector allocation percentages for that profile.
        - Calculates the total market value of the account's holdings.
        - Iterates through each holding:
            - Retrieves the stock's current price.
            - Calculates the total value of that stock holding in the account.

        - For each sector associated with the holding, calculates the weight (percentage) of that sector within the account's total value.
    - Returns a map containing sector names as keys and their corresponding weight percentages as values.

5. System Analysis - Stock Recommendations & Advisor Groups (High-Level):

stockRecommendations(accountId, maxRecommendations, numComparators):

- Initialize an empty map `recommendations` to store stock recommendations (stock symbol as key, buy/sell recommendation as value).
- Initialize an empty map `similarAccounts` to store account IDs and their cosine similarity to the target account.
- Iterate through all accounts (excluding the target account):
    - Calculate cosine similarity between the target account's holdings and the current account's holdings.
    - Add the current account ID and its cosine similarity to `similarAccounts`.
- Sort `similarAccounts` by cosine similarity in descending order (most similar first).
- Initialize a counter `count` to track the number of recommendations made.
- Iterate through the top `numComparators` (or less if there are fewer accounts) similar accounts:
    - For each stock held by the target account:
    - If `count` reaches `maxRecommendations`, break the loop.
    - If the similar account doesn't hold the stock, recommend buying the stock and increment `count`.
    - For each stock held by the similar account:
    - If `count` reaches `maxRecommendations`, break the loop.
    - If the target account doesn't hold the stock, recommend selling the stock and increment `count`.
- Prioritize recommendations based on majority holding and total investment from similar accounts (specific logic omitted for brevity).
- Return the `recommendations` map.

Advisor Grouping using k-means Clustering:

clusterAccounts(List<Account> accounts)

- Create a new `KMeansClustering` object with desired tolerance and maximum number of groups.
- Initialize `clusters` as an empty list to store account groupings.
- Start with 1 cluster:
    - Create a list of `Cluster` objects, each representing a cluster with a unique ID and an initial centroid (average sector weights of all accounts).
- Iterate until convergence (all accounts assigned to closest cluster without changes) or maximum number of groups reached:
    - Loop through all accounts:

    - Calculate the cosine similarity between the account's sector weights and the centroid of each cluster.
    - Assign the account to the cluster with the closest centroid (based on cosine similarity).
    - If the account's assigned cluster changes, set `converged` to false (requires another iteration).
    - If `converged` is false (assignments changed):
    - For each cluster, update its centroid by calculating the average sector weights of all accounts assigned to that cluster.
- Add the final cluster assignments to the `clusters` list (each list within `clusters` represents an advisor group).
- Return the `clusters` list.

Limitations

Account Management:

- **Reinvest Option Update:** Currently, the system lacks a method to update the reinvest option for an existing account. Users should be able to modify this preference after account creation.

Dividend Management:

- **Disbursement Simplification:** The disburseDividend method treats dividend reinvestment as a binary option (reinvest or add to cash). Consider incorporating more sophisticated reinvestment strategies, including fractional share purchases based on tax implications or user-defined preferences.

Investment Profile Management:

- **Dynamic Profile Updates:** The system assumes static investment profiles with fixed sector allocations. Allow users to adjust their sector weightings within their profiles over time based on market changes or risk tolerance modifications.

Data Flexibility:

- **Fixed Sector Weights:** The system might not allow changing sector weights within existing investment profiles. Implement a mechanism for users or advisors to modify sector weightings within profiles.

Test Plan:

Overview of the Black Box Testing Plan

- Defining Test Cases:
    - Create tests for each method covering input validation, boundary cases, control flow, and data flow.
    - Test for unusual but valid inputs and invalid inputs to assess system robustness.
    - Verify error handling according to method specifications.
- Data Entry Testing:
    - Validate data entry methods by ensuring correct storage and retrieval of various data entities.
- Reporting Method Testing:
    - Confirm accuracy of reporting methods such as account value, advisor portfolio value, and investor profit.
- Analysis Method Testing:
    - Evaluate accuracy and reliability of analysis methods like divergent accounts and stock recommendations.
- Integration Testing:
    - Assess interactions between different parts of the system, particularly the impact of methods like tradeShares.

Test Cases:

defineSector (String sectorName)

Test Cases for defineSector Method.

Input Validation:

- Test with null for sectorName.
- Test with an empty string for sectorName.

Boundary Tests:

- Test defining a sector that already exists.

Control Flow:

- Test defining a sector name with special characters (e.g., hyphens, underscores).
- Test defining a sector name with leading and trailing spaces.
- Test defining a sector name with mixed case characters.
- Verify duplicate sector names are rejected.

Data Flow:

- Test consecutive calls to defineSector (should succeed for unique names).
- Test defining a sector after defining a stock (ensure proper association).
- Test defining a sector after trading activities (no impact expected).
- Test defining a sector before any trading activities.
- Test defining a sector after creating a new investment account.
- Test defining a sector after creating a new investment profile.
- Test defining a sector before creating a new investment profile (allowed).
- Test defining a sector after distributing dividends (no impact expected).
- Test defining a sector after setting a new stock price (no impact expected).
- Test defining a sector after adding advisors and clients (allowed).
- Test defining a sector before adding advisors and clients (allowed).

defineStock (String companyName, String stockSymbol, String sectorName)

Test Cases for defineStock Method.

Input Validation:

- Test with null for each parameter (companyName, stockSymbol, sectorName).
- Test with empty strings for each parameter (companyName, stockSymbol, sectorName).

Boundary Tests:

- Test defining a stock with a symbol that already exists.

Control Flow:

- Test defining a stock with a company name containing special characters (e.g., hyphens, underscores).
- Test defining a stock with a company name containing leading and trailing spaces.
- Test defining a stock with a company name containing mixed case characters.
- Verify duplicate stock symbols are rejected (regardless of company name).

Data Flow:

- Test consecutive calls to defineStock with unique symbols (should succeed).
- Test defining a stock after defining a sector (ensure association).
- Test defining a stock after trading activities (no impact expected).
- Test defining a stock before any trading activities.
- Test defining a stock after creating a new investment account (allowed).
- Test defining a stock after creating a new investment profile (allowed).
- Test defining a stock before creating a new investment profile (allowed).
- Test defining a stock after distributing dividends (no impact expected).
- Test defining a stock after setting a new stock price (allowed).
- Test defining a stock after adding advisors and clients (allowed).
- Test defining a stock before adding advisors and clients (allowed).

setStockPrice(String stockSymbol, double perSharePrice)

Input Validation:

- Test with null for stockSymbol. (Unchanged)
- Test with an empty string for stockSymbol. (Unchanged)

Boundary Tests:

- Test setting price for an undefined stock symbol. (Unchanged)
- Test setting price to a negative value. (Unchanged)

Control Flow:

- Test updating price for the same stock multiple times (**High Frequency:** Ensure efficient handling of frequent updates).
- Test setting price after selling the stock (**Trade Simulation:** Simulate selling a stock and update price).
- Test setting price after buying the stock (**Trade Simulation:** Simulate buying a stock and update price).
- Test setting price with a decimal value (e.g., 100.5). (Unchanged)
- Test setting price to zero (**Valid Trade?** Clarify if zero price is a valid trade scenario).

Data Flow:

- Test setting price for a newly defined stock (**Post-Definition Update:** Allow price update after stock definition).
- Test setting price multiple times for the same stock (**Multiple Trades:** Simulate multiple trades for the same stock).
- Test setting price after a stock trade (**Redundancy Check:** Ensure setStockPrice doesn't duplicate trade data).
- **New:** Test setting price concurrently from multiple sources (**Concurrency:** Simulate multiple sources updating the price simultaneously).
- Test setting price after getting total account value (**Possible Dependency:** Check for any dependency between price update and account value).

defineProfile(String profileName, Map sectorHoldings)

Test Cases for defineProfile Method

Input Validation:

- Test with null for profileName.
- Test with an empty string for profileName.
- Test with null for sectorHoldings.
- Test with an empty sectorHoldings map.

Data Flow:

- Test defining a profile with a unique name and valid sectorHoldings (percentages should add up to 100).
- Test defining a profile with a name that already exists (should fail or overwrite depending on requirements).
- Test defining a profile with a missing "cash" sector (ensure default or error handling).
- Test defining a profile with negative holdings for a sector (should be rejected or capped at 0%).
- Test defining a profile with holdings exceeding 100% total (should be rejected or normalized).
- Test defining a profile with sector names containing special characters (e.g., hyphens, underscores).
- Test defining a profile with holdings for undefined sectors (should be ignored or handled appropriately).

Boundary Tests:

- Test defining a profile with a very large number of sectors in the holdings map (performance implications).

Control Flow:

- Test defining a profile and then retrieving it to verify the stored data.
- Test defining a profile and then modifying the sectorHoldings map (ensure changes are not reflected in the profile).
- Test defining a profile and then using it for portfolio allocation (simulate usage scenario).

Additional Considerations:

- Clarify the expected behaviour for defining a profile with a name that already exists (overwrite or rejection).
- Define how to handle undefined sectors in the sectorHoldings map (ignore, error, or default value).

int addAdvisor (String advisorName)

Input Validation:

- Test with null for advisorName.
- Test with an empty string for advisorName.

Data Flow:

- Test adding an advisor with a unique name (successful addition).
- Test adding an advisor with a name that already exists (should fail or create a duplicate depending on requirements).
- Test adding multiple advisors with different names (verify unique IDs).

Output Validation:

- Verify the returned advisor ID is a positive integer.

Int addClient(String clientName)

Input Validation:

- Test with null for clientName.
- Test with an empty string for clientName.

Data Flow:

- Test adding a client with a unique name (successful addition).
- Test adding a client with a name that already exists (should fail or create a duplicate depending on requirements).
- Test adding multiple clients with different names (verify unique IDs).

Output Validation:

- Verify the returned client ID is a positive integer.

Additional Considerations:

- Clarify the expected behaviour for adding an advisor/client with a name that already exists (overwrite or rejection).
- Test adding clients and then associating them with advisors (simulate real-world scenario).

int createAccount(int clientId, int financialAdvisor, String accountName, String profileType, boolean reinvest)

Input Validation:

- Test with an invalid clientId (non-existent client).
- Test with an invalid financialAdvisor ID (non-existent advisor).
- Test with null for accountName.
- Test with an empty string for accountName.
- Test with an invalid profileType (non-existent profile).

Data Flow:

- Test creating an account with valid parameters (successful creation).
- Test creating an account with the same name for the same client (allowed or rejected depending on requirements).
- Test creating an account with a valid reinvest option (true or false).
- Test creating an account and then associating it with a profile (simulate real-world scenario).
- Test creating an account for a client assigned to an advisor (verify advisor association).

Output Validation:

- Verify the returned account ID is a positive integer.

Boundary Tests:

- Test creating an account with a very long accountName (potential truncation issues).

Control Flow:

- Test creating an account and then retrieving it to verify stored data.
- Test creating an account and then depositing/withdrawing funds (ensure functionality).
- Test creating an account and then simulating a trade (ensure proper execution).

Additional Considerations:

- Clarify the behavior for creating an account with a duplicate name for the same client (allowed or rejected).
- Test account creation and association with a profile that doesn't exist yet (allowed or error handling).

tradeShares(int account, String stockSymbol, int sharesExchanged)

Input Validation:

- Test with an invalid account ID (non-existent account).
- Test with an invalid stockSymbol (non-existent stock).
- Test with a zero value for sharesExchanged (no buy or sell).
- Test with negative sharesExchanged for a stock that isn't "cash" (not allowed for buying).

  Buying Stocks:

  - Test buying shares with a positive sharesExchanged for a valid stock (ensure purchase, holding update, and cash decrease).
  - Test buying shares with insufficient funds in the account (expected failure or error handling).
  - Test buying a fractional share quantity (ensure correct handling).
  - Test buying a stock that already exists in the account (shares should be added).

  Selling Stocks:

  - Test selling shares with a negative sharesExchanged for a valid stock (ensure sale, holding update, and cash increase).
  - Test selling shares exceeding the available holdings for that stock (expected failure or error handling).
  - Test selling all shares of a stock owned by the account (stock holding should be removed).

  Cash Transfer:

  - Test transferring cash into the account with a positive sharesExchanged for "cash" stock symbol (ensure cash balance increase).
  - Test transferring cash out of the account with a negative sharesExchanged for "cash" stock symbol (ensure cash balance decrease, not below zero).
  - Test transferring cash exceeding the available balance (allowed for positive transfers, rejected for negative).

Data Flow:

- Test buying and then selling the same stock (ensure correct cash flow and holding changes).
- Test buying and then disbursing dividends with reinvestment enabled (ensure dividends are used for additional purchase).
- Test buying a stock and then verifying the updated account cash balance and holdings.
- Test selling a stock and then verifying the updated account cash balance and holdings.

Control Flow:

- Test trading shares and then verifying the updated account information (cash balance, holdings).
- Test trading shares with a stock price set by setStockPrice (ensure correct price used).
- Test trading shares with no prior price set (default price of $1 applied).

Boundary Conditions:

- Test buying/selling a very large number of shares (ensure system handles large transactions).
- Test transferring cash that would result in a very low cash balance (ensure it doesn't go below zero).

Error Handling:

- Test trading shares with insufficient funds and verify the returned error code/message.
- Test trading shares for a non-existent stock and verify the returned error code/message.

changeAdvisor(int accountId, int newAdvisorId)

Test Cases for changeAdvisor Method:

Input Validation:

- Test with an invalid accountId (non-existent account).
- Test with an invalid newAdvisorId (non-existent advisor).

Data Flow:

- Test changing the advisor for an account with a valid new advisor (successful update).
- Test changing the advisor for an account multiple times (verify the latest advisor is associated).
- Test changing the advisor to the same advisor it currently has (no change expected).

Control Flow:

- Test changing the advisor and then retrieving the account information to verify the updated advisor ID.
- Test changing the advisor and then simulating a trade (ensure new advisor association is considered).

Additional Considerations:

- Clarify the behavior if the newAdvisorId belongs to an already assigned advisor for the client (allowed or rejected).

double accountValue(int accountId)

Input Validation:

- Test with an invalid accountId (non-existent account).

Data Flow:

- Test calculating the account value with various stock holdings and a cash balance (ensure correct calculation).
- Test calculating the account value with no stock holdings (should return just the cash balance).
- Test calculating the account value with fractional shares (ensure correct handling).
- Test calculating the account value after buying stocks (reflected in market value).
- Test calculating the account value after selling stocks (reflected in market value).
- Test calculating the account value after depositing cash (reflected in market value).
- Test calculating the account value after withdrawing cash (reflected in market value).

Control Flow:

- Test calculating the account value and then verifying the returned value against manual calculations.
- Test calculating the account value before and after a trade (ensure market value reflects the trade).

double advisorPortfolioValue(int advisorId)

Input Validation:

- Test with an invalid advisorId (non-existent advisor).

Data Flow:

- Test calculating the advisor's portfolio value with one client account (ensure correct total value).
- Test calculating the advisor's portfolio value with multiple client accounts (ensure sum of account values).
- Test calculating the portfolio value after a client opens a new account (reflected in total value).
- Test calculating the portfolio value after a client closes an account (removed from total value).
- Test calculating the portfolio value after a client's account value changes (reflected in total value).

Control Flow:

- Test calculating the advisor's portfolio value and then verifying the returned value against manual calculations (considering all client accounts).
- Test calculating the portfolio value before and after a trade in one of the client's accounts (ensure advisor's total value reflects the change).

Additional Considerations:

- Consider testing the behaviour for an advisor with no client accounts (should return 0 or handle gracefully).
- These test cases assume the methods use current stock prices. If historical prices are used, additional test cases might be needed to cover that scenario.

Map investorProfit(int clientId)

Input Validation:

- Test with an invalid clientId (non-existent client).

Data Flow:

- Test calculating the investor profit with one account and one stock holding (ensure correct calculation based on ACB).
- Test calculating the investor profit with one account and multiple stock holdings (ensure sum of individual profits).
- Test calculating the investor profit with multiple accounts and various stock holdings (ensure total profit across all accounts).
- Test calculating the profit with fractional shares (ensure correct handling in ACB calculation).
- Test calculating the profit with no stock holdings in any account (should return 0).
- Test calculating the profit after buying stocks at different prices (ACB should reflect average cost).
- Test calculating the profit after selling some shares of a stock (ACB should be adjusted proportionally).

Control Flow:

- Test calculating the investor profit and then verifying the returned value against manual calculations (considering all accounts and holdings).
- Test calculating the profit before and after buying/selling stocks (ensure profit reflects the changes).

Boundary Conditions:

- Test calculating the profit with a large number of stock holdings (ensure system handles complex calculations).

Additional Considerations:

- Cash balances are explicitly excluded from the profit calculation. Consider testing to ensure cash is not included.
- Consider testing the behavior for a client with no investment accounts (should return 0 or handle gracefully).

Map profileSectorWeights(int accountId)

Input Validation:

- Test with an invalid accountId (non-existent account).

Data Flow:

- Test calculating sector weights with one account and one stock holding (ensure correct weight based on market value).
- Test calculating sector weights with one account and multiple stock holdings in different sectors (ensure sum of weights is 100).
- Test calculating sector weights with multiple accounts and various stock holdings (ensure weights for each account reflect sector distribution).
- Test calculating sector weights with fractional shares (ensure correct handling in market value calculation).
- Test calculating sector weights with an account holding cash (cash shouldn't be included in sector weights).
- Test calculating sector weights after buying stocks in a particular sector (reflected in weight increase).
- Test calculating sector weights after selling stocks from a particular sector (reflected in weight decrease).

Control Flow:

- Test calculating sector weights and then verifying the returned map against manual calculations (considering all holdings and sector associations).
- Test calculating sector weights before and after a trade (ensure weights reflect the changes in holdings).

Boundary Conditions:

- Test calculating sector weights with a very large number of stock holdings (ensure system handles complex calculations).

Additional Considerations:

- The test cases assume current stock prices are used for market value calculation. If historical prices are used, additional test cases might be needed.
- Testing the behavior for an account with no stock holdings (should return a map with all sectors at 0% or handle gracefully).

- Test if the method includes sectors with no holdings (0% weight) or excludes them from the returned map.

Set divergentAccounts(int tolerance)

Input Validation:

- Test with a negative tolerance value (not allowed).

Data Flow:

- Test identifying divergent accounts with holdings slightly outside the tolerance (ensure correct identification).
- Test identifying divergent accounts with holdings significantly outside the tolerance (ensure correct identification).
- Test identifying divergent accounts with cash holdings outside the tolerance range (ensure cash is considered).
- Test identifying divergent accounts with multiple sectors exceeding the tolerance (ensure all are reported).
- Test identifying divergent accounts with no sectors exceeding the tolerance (should not be reported).
- Test identifying divergent accounts after buying stocks in a particular sector (may cause divergence if outside tolerance).
- Test identifying divergent accounts after selling stocks from a particular sector (may cause divergence if outside tolerance).

Control Flow:

- Test identifying divergent accounts and then verifying the returned list against manual calculations (considering all accounts and sector weights).
- Test calculating sector weights and then calling divergentAccounts to ensure consistent results.
- Test identifying divergent accounts before and after a trade (ensure list reflects changes in holdings).

Boundary Conditions:

- Test identifying divergent accounts with a very large number of accounts (ensure system handles complex calculations).
- Test identifying divergent accounts with a tolerance value of 0 (all accounts will be divergent).

Additional Considerations:

- The test cases assume current stock prices are used for market value calculation. If historical prices are used, additional test cases might be needed.

- Consider testing the behavior for accounts with no stock holdings (how are they handled in divergence calculation?).

int disburseDividend(String stockSymbol, double dividendPerShare)

Input Validation:

- Test with an invalid stockSymbol (non-existent stock).
- Test with a negative dividendPerShare value (not allowed).

Data Flow:

- Test disbursing dividends for a stock with no account holdings (no impact expected).
- Test disbursing dividends with reinvestment disabled (ensure dividends added to cash balance).
- Test disbursing dividends with reinvestment enabled for an account with integer shares (ensure correct purchase and cash decrease).
- Test disbursing dividends with reinvestment enabled for an account with fractional shares (ensure fractional share purchase and cash decrease).
- Test disbursing dividends with reinvestment enabled for multiple accounts with various fractional share needs (ensure firm tracks ownership correctly).
- Test disbursing dividends for multiple accounts with the same stock (ensure separate tracking for each account).
- Test disbursing dividends after a previous disbursement for the same stock (ensure firm's ownership is considered).

Control Flow:

- Test disbursing dividends and then verifying the updated account cash balance and holdings (including firm-owned fractions).
- Test disbursing dividends with a stock price set by setStockPrice (ensure correct price used for purchase).
- Test disbursing dividends for an account with insufficient cash for reinvestment (ensure no purchase and potential error handling).

Boundary Conditions:

- Test disbursing dividends with a very small dividendPerShare (ensure fractional share calculations handle small values).
- Test disbursing dividends with a very large number of accounts holding the stock (ensure system handles complex calculations).

Additional Considerations:

- Consider testing the behavior for a stock with no previous firm holdings (firm acquires new shares for reinvestment).

- Consider testing how the firm manages its fractional share ownership over time (e.g., selling accumulated fractions when possible).
- Define how rounding of fractional shares below 0.0001 should be handled (test cases can verify the implementation).

Map stockRecommendations(int accountId, int maxRecommendations, int numComparators)

Test Cases:

- **Input Validation:**
    - Test with negative accountId, maxRecommendations, or numComparators (not allowed).
- **Data Flow:**
    - Test recommendations with an account holding no stocks (no recommendations expected).
    - Test recommendations with an account holding stocks:
        - Recommend selling a stock if similar accounts mostly don't hold it.
        - Recommend buying a stock if similar accounts mostly hold it.
    - Test recommendations with tied majority:
        - Prioritize buying stocks with higher total investment in similar accounts.
        - Prioritize selling stocks where the account has the biggest holding.
    - Test recommendations with various maxRecommendations (ensure correct number returned).
- **Control Flow:**
    - Test recommendations and verify suggested buys/sells against manually calculated recommendations based on mock data.
    - Simulate changes in account holdings and compare recommendations before and after.

Additional Considerations:

- How are stocks with equal recommendation strength handled?

Set<Set> advisorGroups(double tolerance, int maxGroups)

The advisorGroups method uses k-means clustering to identify groups of financial advisors with similar investment preferences based on sector weight differences. Here's how we can approach testing it:

Overall Functionality of advisorGroups:

Input Validation:

Test with negative tolerance or maxGroups (not allowed).

Data Flow:

Test with a small number of advisors with diverse preferences (ensure multiple clusters).

Test with advisors having similar preferences (ensure they are grouped together).

Test with an advisor managing accounts in multiple clusters (ensure advisor appears in both groups).

Test with the number of clusters reaching maxGroups (ensure the algorithm terminates).

Control Flow:

Run the method with different maxGroups values and compare the returned advisor groupings.

Modify advisor sector weight differences and verify how the groupings change.

Integration with Other Methods:

- Consider testing how advisorGroups works with the profileSectorWeights method to ensure accurate sector weight difference calculations.

Additional Considerations:

- The test cases assumes the cosine similarity calculation is correct.
- How are empty clusters handled (if a cluster has no advisors after iterations)?
- Consider testing the performance of the method with a large number of advisors (ensure it scales efficiently).