STAT295
Object Oriented Programming

INVESTMENT MANAGEMENT SYSTEM
JAVA PROJECT

FURKAN ÇELIK - 2643609
SADIK DINLEMEZ - 2561207
ENES YANAC - 2561587
ABULFAZ KHALILOV - 2655991

Investment Management System

Final Version Report

The Investment Management System (IMS) is a comprehensive, JavaFX-based desktop application designed to revolutionize how diverse financial stakeholders interact with investment data and manage financial assets. Addressing the common challenges of fragmented tools and information silos, the IMS provides a unified, role-centric platform for Normal Users (individual investors), Economists, Financial Analysts, and Portfolio Managers. Its core objective is to enhance decision-making, streamline workflows, and foster collaboration by delivering tailored functionalities and data-driven insights within an intuitive graphical environment.

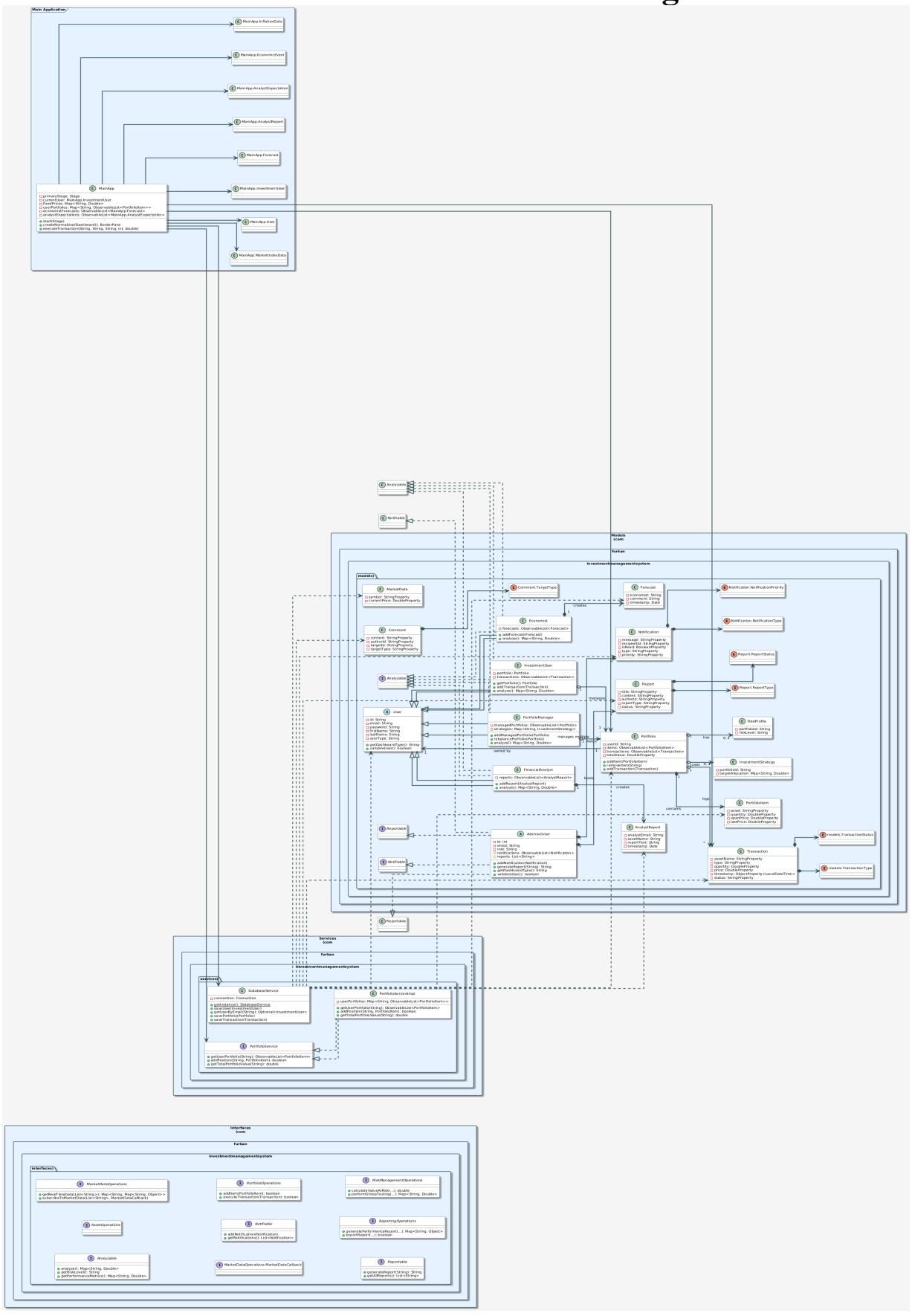
For Normal Users, the IMS offers robust personal portfolio management, including real-time (simulated) tracking of stocks, cryptocurrencies, and forex assets, transaction execution, performance visualization through dynamic charts, and a direct communication channel for receiving advice from Portfolio Managers. Economists are equipped with tools to formulate and disseminate macroeconomic forecasts (e.g., interest rates, inflation, currency movements), contributing to a shared pool of economic intelligence. Financial Analysts can conduct in-depth asset analysis, generate detailed research reports with buy/sell/hold recommendations, and share their findings system-wide. Finally, Portfolio Managers utilize a powerful dashboard to oversee multiple client portfolios, synthesize economic and analytical insights, provide personalized client advice, and make strategic portfolio adjustments.

Architecturally, the IMS is built on sound object-oriented principles, featuring a clear separation of concerns into distinct layers: detailed Models representing financial entities (User, Portfolio, Transaction, etc.) with JavaFX property bindings for UI reactivity; Interfaces defining contracts for core operations (portfolio management, market data handling, reporting, risk analysis); and a Service Layer encapsulating business logic and data persistence (initially in-memory for portfolio services, with a DatabaseService for SQLite integration). The MainApp class orchestrates the user interface, dynamically rendering role-specific dashboards rich with interactive tables, charts, and input forms.

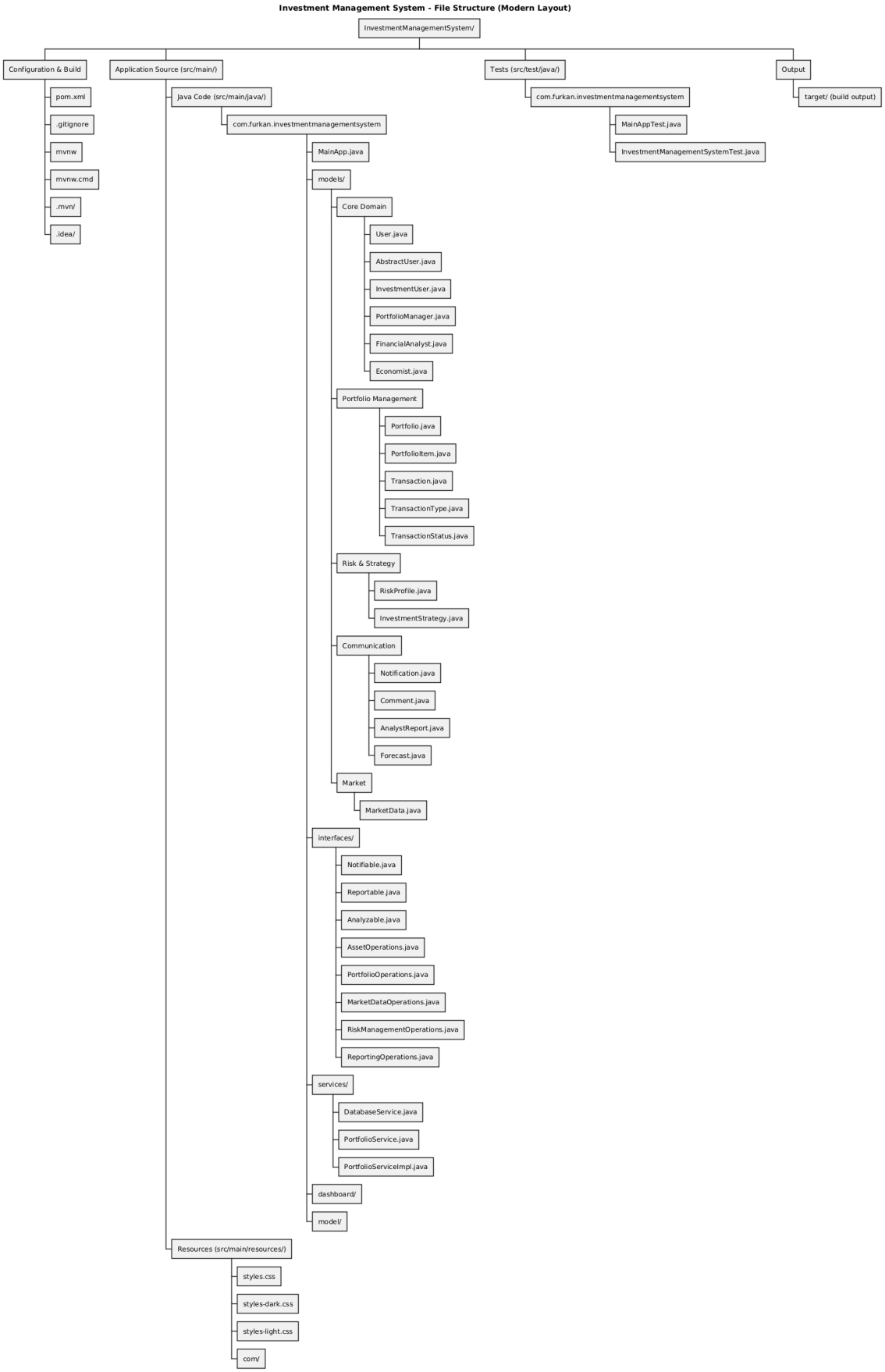
The system's design emphasizes modularity and extensibility, facilitated by its interface-driven approach. This not only ensures maintainability but also paves the way for future enhancements such as live API integration for market data, advanced analytics, and broader platform accessibility. Rigorous unit testing, employing JUnit 5 and Mockito, validates both the interface contracts and the correctness of core model and service logic, ensuring system reliability.

In essence, the Investment Management System provides a sophisticated yet user-friendly solution that centralizes investment activities, empowers users with relevant tools and information, and lays a robust foundation for future growth and innovation in financial technology. **A detailed explanation of the project's architecture, components, functionalities, and design principles is provided in the subsequent sections of this report.**

Final Version UML Diagram



Final Version Software Architecture



Investment Management System

Detailed Final Version Report

I. Introduction and Project Overview

The Investment Management System is a comprehensive JavaFX desktop application meticulously designed to serve as a versatile platform for individuals and professionals operating within the financial investment landscape. Its core purpose is to streamline and enhance various aspects of investment management, from personal portfolio tracking to sophisticated institutional analysis and decision-making.

- **Target Users & Scope**: The system caters to a diverse range of user roles:
  - **Normal Users (Individual Investors)**: Empowered to manage their personal investment portfolios, execute buy/sell transactions, monitor asset performance, and gain insights into their financial standing.
  - **Economists**: Provided with tools to analyze macroeconomic trends, formulate forecasts on key economic indicators (like interest rates, inflation, currency movements), and disseminate these insights within the system.
  - **Financial Analysts**: Equipped to conduct in-depth analysis of specific assets, generate detailed research reports, provide investment recommendations (buy, sell, hold), and share their findings.
  - **Portfolio Managers**: Given an oversight role to manage multiple client portfolios, synthesize information from economists and analysts, provide tailored advice to clients, and make strategic adjustments to managed portfolios.
- **Key Objectives & Benefits**:
  - **Centralized Platform**: Offers a single, integrated environment for various investment-related activities, reducing the need for disparate tools.
  - **Role-Specific Functionality**: Delivers a tailored user experience and feature set for each user type, ensuring relevance and efficiency.
  - **Data-Driven Insights**: Facilitates informed decision-making through data visualization (charts, tables), performance metrics, and analytical reports.
  - **Improved Workflow**: Streamlines processes like transaction execution, report generation, and communication of financial insights between different roles.
  - **Modularity and Extensibility**: Designed with a focus on clear separation of concerns and interface-based programming to allow for easier maintenance and future enhancements.

The system architecture emphasizes a robust model-view-controller (MVC)-like pattern (though JavaFX often blends Controller and View aspects), with distinct layers for data models, service logic, and user interface presentation. It leverages JavaFX for a rich graphical user interface and incorporates object-oriented principles to create a well-structured and maintainable application.

II. System Architecture: Classes and Components

The system's codebase is organized into logical packages, primarily models, interfaces, services, and the main application logic residing in the root package.

A. Model Classes (com.furkan.investmentmanagementsystem.models and MainApp inner classes)

These classes are the backbone of the application, representing the data entities and their inherent behaviors. They utilize JavaFX properties (StringProperty, DoubleProperty, etc.) extensively to enable easy binding with the UI, ensuring that changes in the data model are automatically reflected in the view and vice-versa.

1. **User (abstract - models package)**
   - **Purpose**: Serves as the fundamental, abstract blueprint for all user types within the system. It defines common attributes and behaviors expected of any user.

- **Key Attributes**: id (String), email (String), password (String), firstName (String), lastName (String), userType (String).
- **Key Methods**:
  - getId(), getEmail(), getPassword(), getFirstName(), setFirstName(), getLastName(), setLastName(), getUserType(), getFullName(): Standard getters and setters for user attributes.
  - getDashboardType() (abstract): Must be implemented by concrete subclasses to specify which dashboard UI is relevant for that user role.
  - validateUser() (abstract): Must be implemented by subclasses to define the logic for authenticating the user (e.g., checking credentials).

2. **AbstractUser (abstract - models package)**
   - **Purpose**: Provides a more feature-rich abstract base for users, specifically incorporating notification and reporting capabilities by implementing the Notifiable and Reportable interfaces. This class seems to offer an alternative or supplementary user hierarchy.
   - **Key Attributes**: id (int), email (String), role (String), password (String), firstName, lastName, notifications (ObservableList<Notification>), reports (List<String>), lastLogin (LocalDateTime), isActive (boolean).
   - **Key Methods**:
     - Implements all methods from Notifiable (e.g., addNotification, getNotifications, getUnreadNotificationCount).
     - Implements all methods from Reportable (e.g., generateReport, getAllReports).
     - updateLastLogin(), activate(), deactivate(): Manages user activity status.
     - getDashboardType() (abstract), validateUser() (abstract): Similar to the User class.
3. **Concrete User Subclasses (extending User)**:
   - **InvestmentUser**: Represents an individual investor who manages a personal portfolio.
     - **Key Attributes**: Inherits from User, portfolio (Portfolio), transactions (ObservableList<Transaction>), totalAssets (DoubleProperty), riskTolerance (StringProperty), investmentStrategy (StringProperty).
     - **Key Methods**:
       - getPortfolio(): Returns the user's investment portfolio.
       - addTransaction(Transaction transaction): Adds a new transaction and updates the portfolio accordingly.
       - analyze(): Implements Analyzable to provide metrics like total assets, portfolio item count.
       - getRiskLevel(): Returns the user's self-defined risk tolerance.
       - getDashboardType(): Returns "NormalUser".
   - **Economist**: Represents a user providing economic analyses and forecasts.
     - **Key Attributes**: Inherits from User, forecasts (ObservableList<Forecast>), economicIndicators (Map<String, Double>).
     - **Key Methods**:
       - addForecast(Forecast forecast): Adds a new economic forecast.
       - getForecasts(): Retrieves all forecasts made by this economist.
       - analyze(): Implements Analyzable to provide metrics based on forecasts and indicators.
       - getDashboardType(): Returns "Economist".
   - **FinancialAnalyst**: Represents a user conducting asset research and generating reports.
     - **Key Attributes**: Inherits from User, reports (ObservableList<AnalystReport>), assetMetrics (Map for sentiment/confidence), assetRecommendations (Map).
     - **Key Methods**:
       - addReport(AnalystReport report): Adds a new analyst report and updates internal metrics.
       - getReports(): Retrieves all reports by this analyst.
       - analyze(): Implements Analyzable for metrics like report count, asset coverage.
       - getDashboardType(): Returns "FinancialAnalyst".

- **PortfolioManager**: Represents a user overseeing multiple client portfolios and strategies.
  - **Key Attributes**: Inherits from User, userReports (Map<String, String> for client-specific notes), portfolioRecommendations (Map), managedPortfolios (ObservableList<Portfolio>), strategies (Map<String, InvestmentStrategy>).
  - **Key Methods**:
    - addUserReport(String userEmail, String report): Adds a report/note for a specific client.
    - addManagedPortfolio(Portfolio portfolio): Takes a client portfolio under management.
    - rebalancePortfolio(Portfolio portfolio): Initiates rebalancing based on the portfolio's strategy.
    - analyze(): Implements Analyzable for metrics like number of users managed, average risk scores.
    - getDashboardType(): Returns "PortfolioManager".

4. **Core Financial Models**:
   - **Portfolio**: Represents a collection of investment assets.
     - **Key Attributes**: id (IntegerProperty), ownerId (IntegerProperty, linking to a User), name (StringProperty), items (ObservableList<PortfolioItem>), transactions (ObservableList<Transaction>), totalValue (DoubleProperty, dynamically calculated), totalRevenue (DoubleProperty, dynamically calculated), riskProfile (ObjectProperty<RiskProfile enum>), strategy (StringProperty).
     - **Key Methods**:
       - addItem(PortfolioItem item): Adds an asset to the portfolio or updates quantity if it exists.
       - removeItem(String assetName): Removes an asset from the portfolio.
       - getItem(String assetName): Retrieves a specific asset.
       - updateItemPrice(String assetName, double newPrice): Updates the price of an asset, triggering recalculations.
       - getAssetTypeDistribution(): Calculates the percentage value of each asset type (Stock, Crypto, Forex) in the portfolio.
       - addTransaction(Transaction transaction): Records a transaction associated with this portfolio.
   - **PortfolioItem**: Represents a single holding within a portfolio.
     - **Key Attributes**: asset (StringProperty, e.g., "AAPL"), type (StringProperty, e.g., "Stock"), quantity (DoubleProperty), openPrice (DoubleProperty), lastPrice (DoubleProperty), totalValue (DoubleProperty, calculated as quantity * lastPrice), totalRevenue (DoubleProperty, calculated profit/loss).
     - **Key Methods**:
       - getAsset(), getType(), getQuantity(), getOpenPrice(), getLastPrice(): Getters for properties.
       - setQuantity(double quantity), setLastPrice(double price): Setters that update properties.
       - getProfitLossPercent(): Calculates the percentage gain or loss.
       - refreshPrice(double newPrice): Updates the last price and recalculates derived values.
       - simulatePriceChange(): Used for mock price updates.
   - **Transaction**: Represents a buy or sell operation.
     - **Key Attributes**: id (StringProperty, unique identifier), assetName (StringProperty), type (StringProperty, storing TransactionType enum's display name), quantity (DoubleProperty), price (DoubleProperty), totalAmount (DoubleProperty, calculated as quantity * price), timestamp (ObjectProperty<LocalDateTime>), status (StringProperty, storing TransactionStatus enum name).
     - **Inner Enums**:

        - TransactionType: BUY, SELL, DIVIDEND, INTEREST, TRANSFER (with display names and colors).
        - TransactionStatus: PENDING, COMPLETED, CANCELLED, FAILED.
    - **Key Methods**: Standard getters and property methods. setStatus(TransactionStatus status) to update transaction state.
5. **Supporting Models**:
    - **Forecast (models package)**: Represents an economic forecast (interest rate, inflation, currency) by an economist.
        - **Key Attributes**: economist (String), interestRate (String), inflation (String), usdTry (String), comment (String), timestamp (Date).
    - **AnalystReport (models package)**: Encapsulates a financial analyst's report on an asset.
        - **Key Attributes**: analystEmail (String), assetName (String), outlook (String, e.g., "Bullish"), reportText (String), timestamp (Date).
    - **Notification**: Represents system alerts or messages for users.
        - **Key Attributes**: id (StringProperty), title (StringProperty), message (StringProperty), recipientId (StringProperty), type (StringProperty, from NotificationType enum), isRead (BooleanProperty).
        - **Inner Enums**: NotificationType, NotificationPriority.
        - **Key Methods**: markAsRead(), delete().
    - **Report**: A generic model for various types of system-generated or user-generated reports.
        - **Key Attributes**: id (StringProperty), title, content, authorId, reportType (from ReportType enum), status (from ReportStatus enum).
        - **Inner Enums**: ReportType, ReportStatus.
        - **Key Methods**: publish(), archive().
    - **Comment**: Represents user comments on entities like reports or portfolio items.
        - **Key Attributes**: id, content, authorId, targetType (from TargetType enum), targetId.
        - **Inner Enum**: TargetType.
        - **Key Methods**: like(), delete().
    - **MarketData**: Stores real-time or historical market information for an asset.
        - **Key Attributes**: symbol, name, assetType, currentPrice, openPrice, volume, marketCap, lastUpdated.
        - **Key Methods**: updatePrice(double newPrice), updateVolume(double newVolume).
    - **InvestmentStrategy**: Details the investment approach for a portfolio.
        - **Key Attributes**: portfolioId, targetAllocation (Map<String, Double> for asset classes), rebalancingThreshold (double), strategyType (String).
        - **Key Methods**: needsRebalancing(Map<String, Double> currentAllocation) checks if the portfolio deviates significantly from its target.
    - **RiskProfile**: Defines risk parameters and limits for a portfolio.
        - **Key Attributes**: portfolioId, riskLevel (String), maxDrawdownLimit (double), assetClassLimits (Map).
        - **Key Methods**: isWithinLimits(Map<String, Double> currentAllocation) checks if allocations adhere to defined limits.
6. **MainApp Inner Model Classes**:
    - User, InvestmentUser (MainApp version), Forecast (MainApp version), AnalystReport (MainApp version), AnalystExpectation, EconomicEvent, InflationData, MarketIndexData: These are simpler, often immutable, data structures primarily used for direct UI display within MainApp. They are distinct from the more comprehensive models in the com.furkan.investmentmanagementsystem.models package which have more behavior and JavaFX properties for robust binding. This separation suggests these inner classes are tailored for transient UI state or simpler data passing to UI components.

B. Interface Definitions (com.furkan.investmentmanagementsystem.interfaces)

Interfaces are blueprints that define a contract of methods without providing implementations. They are crucial for achieving abstraction and polymorphism, allowing different parts of the system to interact in a standardized way.

- **Analyzable**:
  - **Responsibility**: Defines a standard way to perform analysis on various entities.
  - **Key Methods**:
    - analyze(): Returns a Map<String, Double> containing key-value pairs of analysis metrics specific to the implementing entity.
    - getRiskLevel(): Returns a string representing the assessed risk (e.g., "LOW", "MEDIUM", "HIGH").
    - getPerformanceMetrics(): Returns a map of performance-related metrics.
    - validateCriteria(Map<String, Object> criteria): Checks if the entity meets certain predefined criteria.
- **AssetOperations**:
  - **Responsibility**: Specifies a common set of operations that can be performed on any financial asset.
  - **Key Methods**: getCurrentPrice(), getHistoricalPrices(startDate, endDate), getTradingVolume(startDate, endDate), getVolatility(period), getMarketCap(), getDividendYield(), getSector(), isTradable().
- **MarketDataOperations**:
  - **Responsibility**: Defines how market data (real-time, historical, news, etc.) is retrieved and managed.
  - **Key Methods**: getRealTimeData(assetSymbols), getHistoricalData(assetSymbol, startDate, endDate, interval), getMarketIndices(indexSymbols), getMarketNews(assetSymbols, maxResults), subscribeToMarketData(assetSymbols, callback).
  - **Inner Interface**: MarketDataCallback (with onMarketDataUpdate and onError) for handling asynchronous real-time data updates.
- **Notifiable**:
  - **Responsibility**: Allows entities to receive, manage, and display notifications.
  - **Key Methods**: addNotification(Notification notification), removeNotification(String notificationId), getNotifications(), markNotificationAsRead(String notificationId), getUnreadNotificationCount().
- **PortfolioOperations**:
  - **Responsibility**: Standardizes operations performed on an investment portfolio.
  - **Key Methods**: addItem(PortfolioItem item), removeItem(String assetName), updateItemQuantity(assetName, newQuantity), updateItemPrice(assetName, newPrice), executeTransaction(Transaction transaction), getTotalValue(), getAssetTypeDistribution(), rebalance(Map<String, Double> targetAllocations).
- **Reportable**:
  - **Responsibility**: Defines how entities can generate and manage basic reports about themselves.
  - **Key Methods**: generateReport(String entityId), getAllReports(), addReport(String report), removeReport(String reportId).
- **ReportingOperations**:
  - **Responsibility**: Outlines a comprehensive suite of methods for generating various specialized financial reports.
  - **Key Methods**: generatePerformanceReport(...), generateAllocationReport(...), generateTransactionReport(...), generateTaxReport(...), exportReport(reportData, format, filePath), scheduleReport(...).
- **RiskManagementOperations**:
  - **Responsibility**: Specifies a contract for performing detailed risk calculations and management tasks for a portfolio.
  - **Key Methods**: calculateValueAtRisk(confidenceLevel, timeHorizon), calculateExpectedShortfall(...), calculateBeta(marketIndex, lookbackPeriod),

performStressTesting(List<Map<String, Object>> scenarios), setRiskLimits(Map<String, Double> limits), isWithinRiskLimits().

C. Service Layer (com.furkan.investmentmanagementsystem.services)

This layer contains classes that implement the business logic, data processing, and data persistence for the application.

- **DatabaseService**:
  - **Purpose**: Acts as a centralized data access object (DAO) for all interactions with the SQLite database. It implements the Singleton pattern to ensure only one instance handles database connections.
  - **Technology**: Uses SQLite, a lightweight, file-based relational database, suitable for desktop applications. JDBC (Java Database Connectivity) is used for communication.
  - **Key Responsibilities & Methods**:
    - getInstance(): Provides global access to the single DatabaseService instance.
    - initializeDatabase(): Establishes the connection to the SQLite database file.
    - createTables(): Executes SQL DDL (Data Definition Language) statements to create the necessary tables (users, portfolios, portfolio_items, transactions, reports, comments, notifications, market_data) if they don't already exist. Each table schema is designed to store attributes of the corresponding model classes.
    - **Entity-Specific Operations**: For each major model (User, Portfolio, Transaction, etc.), it provides:
      - save[EntityName]([EntityName] entity): Inserts a new record or updates an existing one (INSERT OR REPLACE).
      - get[EntityName]ByCriteria: Retrieves one or more entities based on specified criteria (e.g., getUserByEmail, getUserPortfolios).
      - getAll[EntityName]s(): Fetches all records for an entity type.
      - map[EntityName]FromResultSet(ResultSet rs): A private helper method to convert a row from a ResultSet into an instance of the corresponding model object.
    - close(): Closes the database connection.
  - **Error Handling**: Uses java.util.logging.Logger to log SQL exceptions and other errors.
- **PortfolioService (interface)**:
  - **Purpose**: Defines a clear contract for operations specifically related to managing user investment portfolios. This promotes separation of concerns, as components that need portfolio functionality can depend on this interface rather than a concrete implementation.
  - **Key Methods**:
    - getUserPortfolio(String userEmail): Retrieves the portfolio items for a specific user.
    - addPosition(String userEmail, PortfolioItem item): Adds a new asset holding.
    - updatePosition(String userEmail, String asset, int newQuantity): Modifies the quantity of an existing asset.
    - removePosition(String userEmail, String asset): Deletes an asset holding.
    - findPosition(String userEmail, String asset): Locates a specific asset in the portfolio.
    - getTotalPortfolioValue(String userEmail): Calculates the current total market value.
    - getTotalProfitLoss(String userEmail): Calculates the overall profit or loss.
    - refreshPortfolioPrices(String userEmail): Triggers an update of all asset prices in the portfolio (likely by fetching latest market data).
- **PortfolioServiceImpl**:
  - **Purpose**: Provides a concrete, in-memory implementation of the PortfolioService interface. This is useful for rapid development, testing, or scenarios where a persistent database for portfolios might not be immediately required or is handled elsewhere.
  - **Implementation Detail**: Uses a ConcurrentHashMap<String, ObservableList<PortfolioItem>> userPortfolios to store portfolios. The key is the user's email,

and the value is an ObservableList of their PortfolioItems. ConcurrentHashMap is chosen for thread-safety if the service were to be accessed by multiple threads, although in a typical JavaFX client application, UI updates usually happen on the JavaFX Application Thread.

- **Key Methods**:
  - Implements all methods defined in PortfolioService. For example, addPosition will retrieve the user's ObservableList, then either add a new PortfolioItem or update the quantity of an existing one.
  - getAssetTypeDistribution(String userEmail): Calculates the value distribution of assets by type (e.g., Stock, Crypto) for a user.
  - getPortfolioRiskProfile(String userEmail): Provides a qualitative risk assessment based on the asset type distribution (e.g., "Crypto Heavy" implies higher risk).

D. Main Application Class (MainApp.java)

This class is the heart of the JavaFX application, orchestrating the UI, user interactions, and overall application flow.

- **Role**: It serves as the entry point (main method calls launch(args)), initializes the primary JavaFX Stage, and manages the different Scenes that make up the application's UI.
- **Responsibilities**:
  - **Application Initialization (start() method)**:
    - Sets the title of the primary window.
    - Loads initial data: loadFixedPrices() (populates a map of asset names to their presumed current prices, used for transactions and display), loadMockData() (populates mock economic events, inflation data, etc., for dashboard displays).
    - Initializes UI components that are shared or needed early (e.g., scenes for welcome, login, registration).
    - Starts background tasks like startPriceUpdates() (a Timeline that periodically simulates price changes for assets, demonstrating dynamic UI updates).
  - **Scene Management**: Creates and switches between different scenes (welcomeScene, loginScene, registerScene, mainScene). The mainScene itself is dynamically generated based on the logged-in user's role using methods like createMainSceneForRole(String role).
  - **UI Construction**: Contains numerous private methods (e.g., createNormalUserDashboard(), createEconomistForecastTab(), createAssetAnalysisTab(), createUsersPortfoliosTab()) responsible for building complex UI layouts. These methods typically:
    - Instantiate JavaFX layout panes (VBox, HBox, GridPane, BorderPane, TabPane).
    - Create UI controls (Labels, TextFields, Buttons, ComboBoxes, TableViews, Charts).
    - Configure these controls (set text, prompts, styles, event handlers).
    - Arrange controls within layout panes.
    - Bind UI controls to ObservableLists or JavaFX properties from model objects or MainApp's internal data structures to enable automatic UI updates.
  - **Event Handling**: Attaches event handlers (e.g., setOnAction) to UI controls like buttons and ComboBoxes. These handlers contain the logic to execute when a user interacts with the UI (e.g., attempting login, registering a new user, executing a transaction, saving a forecast).
  - **State Management**: Directly holds and manages various ObservableLists (e.g., loginUsers, investmentUsers, economistForecasts, analystExpectations) and Maps (e.g., fixedPrices, userPortfolios, userTransactions). This data is used to populate UI elements and is modified based on user actions.
    - currentUser: Stores the currently logged-in InvestmentUser (MainApp version).
    - currentUserPortfolioItems & currentUserTransactionItems: ObservableLists specifically for the logged-in normal user's data, enabling reactive UI updates in their dashboard.
  - **Business Logic (Embedded)**: Some business logic, especially related to UI interactions and data manipulation for display, is present directly within MainApp. For example, the

executeTransaction() method in MainApp updates currentUserPortfolioItems and currentUserTransactionItems. While this is common in simpler JavaFX apps, in larger systems, more of this logic might be moved to dedicated service classes.
    - **Styling and Theming**: Applies modern styling constants (MODERN_COLOR_PRIMARY, etc.) and helper methods (applyModernStyle, addFooterToScene) to create a consistent and visually appealing user interface.
    - **Utility Methods**: Contains helper methods like showAlert() for displaying pop-up messages, getAssetCategory() for categorizing assets, and formatting helpers for table columns.

III. Object-Oriented Programming (OOP) Principles in Action

The Investment Management System effectively leverages core OOP principles to create a structured, maintainable, and extensible application.

- **A. Encapsulation**:
    - **Concept**: Bundling data (attributes) and the methods that operate on that data within a single unit (class), and restricting direct access to some of the object's components.
    - **Application**:
        - In PortfolioItem, attributes like asset, quantity, and openPrice are managed as JavaFX properties. While properties offer observability, direct modification from outside is controlled. Methods like setQuantity() or refreshPrice() are provided to modify state, ensuring that any related calculations (like totalValue which is bound) are updated correctly.
        - The Transaction class encapsulates transaction details. The totalAmount is a derived property bound to quantity and price, ensuring it's always consistent without external manual calculation.
        - DatabaseService encapsulates all database connection and query logic. Other parts of the system don't need to know the specifics of SQL queries or connection management; they interact through defined service methods.
    - **Benefits**: Protects data integrity, simplifies class usage (users of the class don't need to know internal details), and makes the system easier to modify (changes within a class are less likely to break other parts of the system if the public interface remains stable).
- **B. Inheritance**:
    - **Concept**: Allows a new class (subclass or derived class) to acquire the properties and methods of an existing class (superclass or base class).
    - **Application**:
        - The models.User class is an abstract base for InvestmentUser, Economist, FinancialAnalyst, and PortfolioManager. These subclasses inherit common attributes like id, email, password, and userType. They also inherit the obligation to implement abstract methods like getDashboardType() and validateUser(), but provide their own specific versions.
        - If AbstractUser were more integrated, concrete user types could inherit its Notifiable and Reportable implementations, promoting code reuse for notification and basic reporting logic.
    - **Benefits**: Promotes code reuse (common attributes and methods are defined once in the superclass), establishes an "is-a" relationship (e.g., an Economist *is a* User), and allows for hierarchical classification of objects.
- **C. Polymorphism**:
    - **Concept**: "Many forms." Allows objects of different classes to be treated as objects of a common superclass or interface. The specific method executed depends on the actual type of the object at runtime.
    - **Application**:
        - **Interface Polymorphism**: The Analyzable interface is implemented by InvestmentUser, Economist, FinancialAnalyst, and PortfolioManager.

        - *Scenario*: A method could take a List<Analyzable> as input. When iterating through this list and calling item.analyze(), the JVM dynamically calls the analyze() method specific to the actual object's type (e.g., InvestmentUser.analyze() or Economist.analyze()).
        - *MainApp Usage*: While not explicitly shown with a List<Analyzable>, if a generic analysis component were built, it could leverage this.
    - **Method Overriding (Subtype Polymorphism)**:
        - The getDashboardType() method is defined as abstract in models.User. Each subclass (InvestmentUser, Economist, etc.) provides its own concrete implementation. In MainApp, when createMainSceneForRole(currentUser.getRole()) is called (assuming currentUser.getRole() internally maps to currentUser.getDashboardType()), the system polymorphically determines which dashboard to create based on the specific type of currentUser.
  - **Benefits**: Increases flexibility and extensibility. New classes can be added that implement existing interfaces or extend superclasses, and existing code that works with the interface/superclass type can seamlessly work with these new classes without modification.
- **D. Abstraction**:
  - **Concept**: Hiding complex implementation details and exposing only the essential features of an object or system.
  - **Application**:
    - **Abstract Classes**: The models.User class is abstract. It defines the general concept of a "User" with common attributes and abstract methods (getDashboardType, validateUser) that represent essential functionalities. However, it doesn't provide concrete implementations for these abstract methods, as those details are specific to each user role (subclass).
    - **Interfaces**: Interfaces like PortfolioOperations, AssetOperations, and ReportingOperations define *what* operations can be performed without specifying *how* they are performed. For example, any class implementing PortfolioOperations must provide an addItem method, but the internal logic for adding an item can vary (e.g., in-memory list vs. database call).
        - *MainApp's Test Class*: InvestmentManagementSystemTest uses mocked interfaces (e.g., @Mock private PortfolioOperations portfolioOperations;). This demonstrates interaction with the *abstraction* of portfolio operations, not a specific implementation.
    - **DatabaseService**: Provides an abstraction layer over the raw JDBC calls and SQL queries. Client code calls methods like saveUser(user) instead of dealing with PreparedStatements directly.
  - **Benefits**: Simplifies complex systems by breaking them into manageable conceptual parts, reduces complexity for the users of the abstracted component, and allows internal implementations to change without affecting client code as long as the abstract interface remains the same.

IV. Interfaces: Defining Contracts and Enabling Flexibility

Interfaces are a cornerstone of the system's design, fostering loose coupling and promoting a plugin-like architecture for various functionalities.

- **A. Purpose of Interfaces**:

  Interfaces in Java define a contract or a specification of behavior. They declare methods that implementing classes must provide. This allows the system to:

  - **Achieve Abstraction**: Focus on *what* an object can do rather than *how* it does it.

- **Enable Polymorphism**: Treat objects of different classes that implement the same interface in a uniform way.
- **Promote Loose Coupling**: Components depend on interfaces (abstractions) rather than concrete classes, making it easier to swap implementations or add new ones without affecting other parts of the system.
- **Facilitate Testability**: Interfaces can be easily mocked or stubbed for unit testing, as seen in InvestmentManagementSystemTest.java.

- **B. Key Interfaces and Their Roles (Detailed)**:
  - **Analyzable**:
    - **Role**: Provides a standard mechanism for any class that represents an entity (like a user, portfolio, or even a specific asset) to expose analytical data about itself.
    - **Methods**:
      - analyze(): The core method, expected to return a Map<String, Double> where keys are metric names (e.g., "TotalValue", "RiskScore", "ReportCount") and values are their corresponding numerical values. Each implementing class tailors these metrics to its specific context.
      - getRiskLevel(): Offers a qualitative risk assessment (e.g., "LOW", "HIGH"). This allows for standardized risk reporting across different analyzable entities.
      - getPerformanceMetrics(): Similar to analyze(), but might focus specifically on performance-related numbers.
      - validateCriteria(Map<String, Object> criteria): Allows for checking if an entity meets certain dynamic criteria, useful for compliance checks or rule-based alerting.
  - **AssetOperations**:
    - **Role**: Abstracts the interactions with financial assets. This could be crucial if the system were to integrate with different data providers or types of asset management backends.
    - **Methods**: Covers a wide range of data points for an asset: getCurrentPrice(), getHistoricalPrices(...) (for charting), getTradingVolume(...), getVolatility(...) (a risk measure), getMarketCap() (size indicator), getDividendYield() (for income stocks), getPriceToEarningsRatio() (valuation metric), getSector(), getIndustry(), isTradable(), getMinimumTradeSize().
  - **MarketDataOperations**:
    - **Role**: Decouples the rest of the application from the specific source or mechanism of obtaining market data. One implementation might use a live API, another might use a database, and a third might use mock data for testing.
    - **Methods**:
      - getRealTimeData(...): Fetches current market data for a list of assets.
      - getHistoricalData(...): Retrieves past price/volume data for charting and analysis.
      - getMarketIndices(...): Gets values for major market benchmarks (e.g., S&P 500).
      - getExchangeRates(...): Fetches currency conversion rates.
      - getMarketNews(...), getMarketSentiment(...), getEconomicIndicators(...): Provide broader market context.
      - subscribeToMarketData(...) and unsubscribeFromMarketData(...) along with the MarketDataCallback inner interface, suggest a mechanism for real-time, streaming data updates.
  - **Notifiable**:
    - **Role**: Creates a uniform way for any part of the system that needs to display or manage notifications (like a User profile or a specific Portfolio view) to handle them.
    - **Methods**: Provide CRUD-like operations for notifications: addNotification(...), removeNotification(...), getNotifications(), and state changes: markNotificationAsRead(...), getUnreadNotificationCount().
  - **PortfolioOperations**:

        - **Role**: Defines the essential actions that can be performed on any investment portfolio, regardless of its specific type or how it's stored.
        - **Methods**: Core portfolio management tasks: addItem(...), removeItem(...), updateItemQuantity(...), updateItemPrice(...), executeTransaction(...). Also, analytical methods: getTotalValue(), getAssetTypeDistribution(), getAssetTransactions(...), getPerformanceMetrics(), and a strategic operation: rebalance(...).
    - **Reportable**:
        - **Role**: Offers a simple contract for entities that can generate basic, self-descriptive reports. This is a more lightweight reporting interface than ReportingOperations.
        - **Methods**: generateReport(String entityId) (likely to produce a summary string for the given entity), getAllReports(), addReport(...), removeReport(...).
    - **ReportingOperations**:
        - **Role**: Abstracts a more sophisticated and comprehensive reporting engine. It allows for the generation of various standardized financial reports.
        - **Methods**: Covers different report types: generatePerformanceReport(...), generateAllocationReport(...), generateTransactionReport(...), generateTaxReport(...), generateRiskReport(...), generateComplianceReport(...), generateCustomReport(...). Also includes utility functions like exportReport(...), scheduleReport(...), and template management (getReportTemplates, saveReportTemplate).
    - **RiskManagementOperations**:
        - **Role**: Centralizes the definition of advanced risk analysis and management functions for portfolios.
        - **Methods**: Includes calculations for standard risk metrics: calculateValueAtRisk(...), calculateExpectedShortfall(...) (CVaR), calculateBeta(...), calculateSharpeRatio(...), calculateSortinoRatio(...), calculateMaximumDrawdown(...). Also covers risk control: setRiskLimits(...), isWithinRiskLimits(), getRiskLimitViolations(), and scenario analysis: performStressTesting(...).
- **C. Scenarios Illustrating Interface Usage (Expanded)**:
    1. **Scenario: ReportingOperations for Multiple Report Formats and Destinations**
        - The ReportingOperations interface defines exportReport(Map<String, Object> reportData, String format, String filePath).
        - The system might have multiple classes implementing this:
            - PdfReportingService exports reports to PDF.
            - CsvReportingService exports data to CSV files.
            - EmailReportingService could format the report data into an email body and send it.
        - A user in the MainApp clicks an "Export Report" button. The application, based on user selection or configuration, can instantiate the appropriate service:
        - `// In MainApp or a ReportController

          ReportingOperations reportExporter; if (selectedFormat.equals("PDF")) { reportExporter = new PdfReportingService(); } else if (selectedFormat.equals("CSV")) { reportExporter = new CsvReportingService(); } // ... reportExporter.exportReport(reportDataMap, selectedFormat, chosenFilePath);`

        - This allows easy addition of new export formats (e.g., Excel) by just creating a new class implementing ReportingOperations without changing the calling code.
    2. **Scenario: MarketDataOperations for Real-time and Delayed Data**
        - The MarketDataOperations interface has getRealTimeData(...) and subscribeToMarketData(...).
        - Implementations could be:
            - LiveApiMarketDataProvider: Connects to a WebSocket or API for live, tick-by-tick data.

        - DelayedMarketDataProvider: Fetches data that is, for example, 15 minutes delayed (common for free data feeds).
        - HistoricalDbMarketDataProvider: Simulates "real-time" by replaying historical data from a database for backtesting purposes.
    - The MainApp or a dedicated market data module can be configured to use one of these providers. Components like the asset ticker or portfolio value updater would simply call methods on the MarketDataOperations instance, unaware of the underlying data source's nature. This is extremely useful for different environments (development, testing with mock data, production with live data).
3. **Scenario: Analyzable for a Centralized Analytics Engine**
    - An AnalyticsEngine class could be designed to periodically run analyses on various parts of the system.
    - It could maintain a List<Analyzable> entitiesToAnalyze;. This list could contain InvestmentUser objects (to analyze their portfolio performance), FinancialAnalyst objects (to track their report generation frequency or impact), or even Portfolio objects directly.
    - The engine's main loop:
    - `// In AnalyticsEngine

      for (Analyzable entity : entitiesToAnalyze) { Map<String, Double> metrics = entity.analyze(); String risk = entity.getRiskLevel(); // Store or display these metrics and risk levels System.out.println("Analysis for " + entity.toString() + ": " + metrics + ", Risk: " + risk); }`

      IGNORE_WHEN_COPYING_START

      content_copy download Use code with caution. Java

      IGNORE_WHEN_COPYING_END

    - This demonstrates how diverse objects, each with its unique way of being analyzed, can be processed uniformly through the Analyzable contract.

V. User Dashboards: Tailored Experiences for Different Roles

The MainApp dynamically constructs and presents dashboards based on the logged-in user's role, ensuring that each user sees only relevant information and tools.

- **A. Normal User Dashboard (Individual Investor)**:
    - **Layout**: Typically a BorderPane with a top section for macro data/ticker, and a central TabPane.
    - **Key UI Components & Workflow**:
        - **Top Section**:
            - HBox (macroDataBox): Displays static or periodically updated macro-economic figures (USD/TRY, Gold, Interest Rate, Inflation, S&P 500, BIST100) in styled VBox "cards."
            - ScrollPane (Scrolling Asset Ticker): An HBox within a ScrollPane showing asset symbols and their current prices, animated to scroll horizontally, providing a glance at market movements.
        - **"My Portfolio" Tab**:
            - **Transaction Form (GridPane)**: Allows users to input new transactions. ComboBoxes for Asset Type (Stock, Crypto, Forex), Asset Name (dynamically populated based on Asset Type selection), and Transaction Type (Buy/Sell). A TextField for Quantity. An "Execute Transaction" Button.

            - **Portfolio Table (TableView<PortfolioItem>)**: The main display of current holdings. Columns include Asset, Type, Quantity, Position (Long/Short), Open Price, Last Price, P/L %, Total P/L (USD), Total Value (USD). Prices and P/L figures are color-coded (green for profit, red for loss).
            - **Summary Labels (HBox)**: Labels at the bottom of the table displaying totalPortfolioValueLabel, totalPLLabel, totalPLPercentLabel.
            - **Charts (HBox)**:
                - PieChart (assetTypePieChart): Visualizes the proportion of total portfolio value allocated to different asset types (Stock, Crypto, Forex). Slices are labeled with type and percentage.
                - BarChart (profitLossBarChart): Shows "Unrealized Gains," "Unrealized Losses," and "Net P/L" as separate bars.
                - BarChart (profitLossChart): Displays the P/L percentage for each individual asset in the portfolio as a bar.
        - **"My Transactions" Tab**:
            - TableView<Transaction>: Lists all historical transactions with details like Asset, Type, Position, Transaction Type (Buy/Sell color-coded), Quantity, Open Price, Last Price (at time of view), P/L %, and Date.
        - **"Portfolio Manager Reports" Tab**:
            - TextArea (pmReportsForNormalUserArea): A read-only area where users can view personalized reports, notes, or recommendations sent by their assigned Portfolio Manager.
    - **Interactions**: Users can execute trades, which updates their portfolio table, transaction history, summary labels, and charts in real-time due to JavaFX bindings and explicit UI refresh calls (updateNormalUserUI).
- **B. Economist Dashboard**:
    - **Layout**: VBox root with a top bar (title, logout) and a central TabPane.
    - **Key UI Components & Workflow**:
        - **Top Macro Indicators**: HBox showing key macro figures (Interest Rate, Inflation, USD/TRY, EUR/TRY) similar to the Normal User dashboard but potentially with more detail or different sources.
        - **"Forecast" Tab**:
            - **Forecast Input Grid (GridPane)**: ComboBoxes for Interest Rate, Inflation, and USD/TRY forecasts (options: Increase, Decrease, Stable, often with visual cues like arrows). A TextArea (commentArea) for detailed explanations. A "Save Forecast" Button.
            - **Economic Calendar (VBox with TableView<EconomicEvent>)**: Displays upcoming economic events, their dates, and potential impact levels.
            - **Monthly Inflation Trend (VBox with LineChart)**: Visualizes historical and potentially projected monthly inflation rates.
        - **"Expectations" Tab**:
            - HBox containing multiple PieCharts. Each chart visualizes the aggregated distribution (Increase, Decrease, Stable) of forecasts submitted by *all* economists in the system for a specific indicator (e.g., one pie chart for Interest Rate forecasts, one for Inflation).
        - **"All Forecasts & Comments" Tab**:
            - ListView<String>: Displays a comprehensive list of all forecasts submitted by all economists, including their names, the forecast details, and their comments.
    - **Interactions**: Economists input their forecasts, which are then added to a central list (economistForecasts). This data is used to update the aggregated pie charts on the "Expectations" tab and the list on the "All Forecasts" tab, providing a collective view of economic sentiment.
- **C. FinancialAnalyst Dashboard**:
    - **Layout**: BorderPane with a top title/logout bar and a central TabPane.

- **Key UI Components & Workflow**:
  - **"Asset Analysis" Tab**:
    - **Asset Selection (HBox with VBox for controls)**: ComboBoxes to select Asset Type and then a specific Asset Name (dynamically populated). Another ComboBox for investment suggestion (Buy, Sell, Hold).
    - **Asset Price Trend Chart (LineChart)**: Displays a mock historical price trend for the selected asset to provide context.
    - **Comment Area (TextArea)**: A larger text area for the analyst to write their detailed analysis, rationale for the suggestion, and outlook.
    - "Save" Button to submit the analysis.
  - **"Other Analyst Expectations" Tab**:
    - **SplitPane**: Divides the tab into two sections.
      - **All Analyst Expectations (ListView<String>)**: Displays a list of all expectations/comments submitted by *all* financial analysts for various assets.
      - **Asset Heatmap (GridPane)**: A grid of Labels, each representing an asset. The label shows the asset symbol and a mock daily percentage change. The background color of the label is green for positive change and red for negative, providing a quick visual overview of market movements.
- **Interactions**: Analysts select an asset, view its mock trend, form an opinion, and submit their analysis and recommendation. These submissions are added to a central list (analystExpectations) and become visible in the "Other Analyst Expectations" tab and potentially to Portfolio Managers.

- **D. PortfolioManager Dashboard**:
  - **Layout**: BorderPane with a top bar and a central TabPane.
  - **Key UI Components & Workflow**:
    - **"Users and Portfolios" Tab**:
      - **User Table (TableView<String>)**: Lists all "NormalUser" clients. Columns show User Email and a summary of their Portfolio (e.g., key holdings).
      - **User Selection and Risk (HBox)**: A ComboBox to select a specific user from the table. A Label (riskProfileLabel) dynamically displays the calculated risk profile of the selected user's portfolio (e.g., "Crypto Heavy," "Balanced").
      - **Comment/Suggestion Box (HBox)**: A TextArea (pmCommentArea) for the Portfolio Manager to write personalized comments, advice, or strategy adjustments for the selected user. A "Send/Save" Button stores this comment, making it visible to the Normal User on their dashboard.
    - **"Economist & Analyst Expectations" Tab**:
      - **SplitPane**:
        - **Economist Expectations (VBox with HBox rows of PieCharts)**: Displays the aggregated economist forecasts for Interest Rate, Inflation, and USD/TRY using pie charts, similar to what the Economist sees. This provides the PM with a quick overview of macroeconomic sentiment.
        - **Financial Analyst Expectations (VBox with ListView<String>)**: Shows a detailed, scrollable list of reports and expectations from all financial analysts, allowing the PM to drill down into specific asset analyses.
  - **Interactions**: The Portfolio Manager can monitor client portfolios, assess their risk, and provide personalized advice. They use the synthesized information from economists and analysts to inform their strategies and recommendations. The dashboard acts as a control center for client management and strategic oversight.

VI. Testing Strategy: Ensuring System Robustness

The project incorporates unit testing using JUnit 5 and Mockito to verify the correctness and reliability of its various components.

- **A. Purpose of Testing**:

  Unit tests are designed to isolate and test the smallest functional parts of an application (individual methods or classes). The primary goals are:

  - **Verify Correctness**: Ensure that each unit of code behaves as expected under various conditions.
  - **Prevent Regressions**: Detect unintended side effects or bugs introduced by new changes or refactoring.
  - **Improve Design**: Writing testable code often leads to better, more modular designs.
  - **Documentation**: Tests can serve as a form of executable documentation, illustrating how different parts of the code are intended to be used.
- **B. test/java/com/furkan/investmentmanagementsystem/InvestmentManagementSystemTest.java**:
  - **Focus**: This test class primarily targets the system's business logic as defined by its interfaces and the behavior of its core model classes. It emphasizes testing interactions *through* interfaces, which is a key aspect of verifying the system's architectural design.
  - **Testing Approach**:
    - **Mockito for Mocks**: It uses @Mock annotations to create mock objects for the operational interfaces (e.g., PortfolioOperations, AssetOperations). Mocks are "dummy" objects that simulate the behavior of real dependencies.
    - **Behavior Definition (when(...).thenReturn(...))**: For each test, specific behaviors are defined for the mock objects. For example, when(portfolioOperations.addItem(any(PortfolioItem.class))).thenReturn(true); tells Mockito that whenever the addItem method is called on the mocked portfolioOperations object with any PortfolioItem, it should return true.
    - **Assertion (assertEquals, assertTrue)**: JUnit assertions are used to check if the actual outcome of an operation matches the expected outcome. For instance, assertEquals(1500.0, portfolioOperations.getTotalValue()); checks if the (mocked) total value is as expected.
    - **Interaction Verification (verify(...))**: Mockito's verify() method is used to ensure that certain methods on the mock objects were indeed called as expected during the test execution (e.g., verify(portfolioOperations).addItem(portfolioItem);).
  - **Specifics Tested**:
    - **Portfolio Operations**: Verifies that adding, updating items, executing transactions, and getting portfolio values/distributions behave correctly when interacting with the PortfolioOperations interface.
    - **Asset Operations**: Tests retrieval of current price, historical prices, trading volume, volatility, and fundamental data through the AssetOperations interface.
    - **Market Data Operations**: Checks fetching of real-time data, market indices, exchange rates, and sentiment via the MarketDataOperations interface.
    - **Risk Management Operations**: Validates calculations like VaR, Expected Shortfall, Beta, Sharpe ratio, and the results of stress testing through the RiskManagementOperations interface.
    - **Reporting Operations**: Tests generation of performance, allocation, transaction reports, and functionalities like exporting and scheduling reports via the ReportingOperations interface.
    - **User Model Logic**: Directly tests methods on InvestmentUser, FinancialAnalyst, and Economist model instances, such as validateUser(), getDashboardType(), analyst.addReport(), and economist.addForecast().

    - **Benefits of this Approach**: This method effectively tests the contracts defined by the interfaces. It ensures that if any component interacts with, for example, PortfolioOperations, it can expect certain behaviors, regardless of the underlying concrete implementation of that interface. It's excellent for testing component integration at an abstract level.
- **C. test/java/com/furkan/investmentmanagementsystem/MainAppTest.java**:
    - **Focus**: This test class is geared more towards testing specific model functionalities that are instantiated or heavily used within the MainApp (or would be by services it directly uses), and testing the concrete PortfolioServiceImpl.
    - **Testing Approach**:
        - **Direct Instantiation**: Instead of primarily relying on mocks for the core objects under test, this class often directly instantiates model classes (e.g., new PortfolioItem(...), new Transaction(...)) and service implementations (e.g., new PortfolioServiceImpl()).
        - **State-Based Testing**: Tests often involve creating objects, performing operations on them, and then asserting the state of these objects or the results of calculations.
    - **Specifics Tested**:
        - **PortfolioItem Creation**: Validates constructor logic, ensuring that items are created correctly with valid inputs and that exceptions are thrown for invalid inputs (e.g., invalid asset type, non-positive price).
        - **Transaction Creation & Status**: Verifies that transactions are initialized with correct types, amounts, and default statuses, and that status changes are handled correctly.
        - **PortfolioServiceImpl Functionality**:
            - Tests the full lifecycle of portfolio positions: adding, finding, updating quantities, and removing positions using the concrete service implementation.
            - Asserts the correctness of portfolio-level calculations performed by PortfolioServiceImpl, such as getTotalPortfolioValue(), getTotalProfitLoss(), getAssetTypeDistribution(), and getPortfolioRiskProfile().
        - **Transaction Operations within MainApp context**: (Though tested via PortfolioServiceImpl here, it reflects logic that MainApp would orchestrate) creation of buy/sell transactions, calculation of total transaction amounts, and transaction status workflows.
    - **Benefits of this Approach**: This approach thoroughly tests the internal logic of concrete classes and service implementations. It's crucial for ensuring that the actual business rules and calculations within these components are correct. It complements the interface-based testing by looking "inside" the implementations.

By combining these two testing approaches, the project aims for both architectural soundness (testing interface contracts) and implementation correctness (testing concrete class logic).

VII. Conclusion

The Investment Management System, as detailed, stands as a well-architected and feature-rich application designed to support a variety of financial roles. Its adherence to OOP principles, particularly through the extensive use of interfaces and a clear model structure, provides a solid foundation for a maintainable, scalable, and robust system. The role-specific dashboards offer tailored user experiences, enhancing usability and productivity. The inclusion of a service layer, exemplified by the DatabaseService and PortfolioServiceImpl, effectively separates concerns and manages data operations.

The testing strategy, employing both interface-based mocking and direct instantiation testing, contributes significantly to the system's reliability and the trustworthiness of its financial calculations and operations.

Future Enhancements Could Include:

- **Full Database Integration**: Complete the persistence of all model entities (PortfolioItem, Transaction, Forecast, AnalystReport, etc.) using the DatabaseService to ensure data is saved between sessions for all features.
- **Real-Time API Integration**: Replace mock data and fixed prices with live data feeds from financial APIs (e.g., Alpha Vantage, IEX Cloud, or cryptocurrency exchanges) for MarketDataOperations.
- **Advanced Analytics and AI**: Incorporate more sophisticated analytical tools, risk modeling (e.g., Monte Carlo simulations), and potentially machine learning for predictive insights or automated strategy suggestions.
- **Enhanced Security**: Implement robust authentication, authorization, and data encryption mechanisms, especially if handling sensitive financial data.
- **Web-Based or Mobile Version**: Extend the platform to web or mobile clients for broader accessibility.
- **User Customization**: Allow users to customize dashboard layouts, preferred metrics, and alert thresholds.
- **Inter-Role Collaboration Tools**: More direct messaging or shared workspace features for economists, analysts, and portfolio managers.
- **Comprehensive Reporting Suite**: Fully implement all methods in ReportingOperations with rich report generation capabilities and export options.