# Software Architecture & Design – Sprint 2

Team Members:

Evan Balson – BAL18466416

Maahia Rahman – RAH23614335

Hala Bakhtiar – BAK23592238

Mahbouba Rezaei – REZ23579670

Roehampton University

Course Number: CMP020N207S

Lecturer: Dr Shekoufeh Kolahdouz Rahimi

18/03/2025

# Overview

StockShark is a web application designed to provide users with quick and accessible stock data, powered by real time information fetched from the Yahoo Finance API. Whether you are seasoned investor or just starting to explore the stock market, StockShark offers a streamlined experience to analyze and compare stocks within a two-year timeframe.

## Key Features:

- Effortless stock search: easily find stocks by their company name or ticker symbol.
- Historical data comparison: visualize and compare the performance of multiple stocks over a customizable period, up to two years.
- Real-time data: leverages the Yahoo Finance API to provide up to date stock information.

## Architecture & Design:

StockShark employs clean architecture, a software design that emphasizes separation of concerns and independent from frameworks and external tools. [1]

Clean architecture is a software design pattern aims for:

- Framework independence: allows to easily switch to a different web framework or database without rewriting the heart of StockShark. [1]
- Easy testing: because the business logic is isolated, we can easily write tests to ensure that is works correctly. That leads to a more reliable application. [1]
- User interface independence: the user interface is separate from the core logic. This means we redesign the website without affecting how StockShark processes stock data. [1]
- External API independence: we can change the external API we are using to get data, without impacting the business rules. [1]

## How is this achieved?

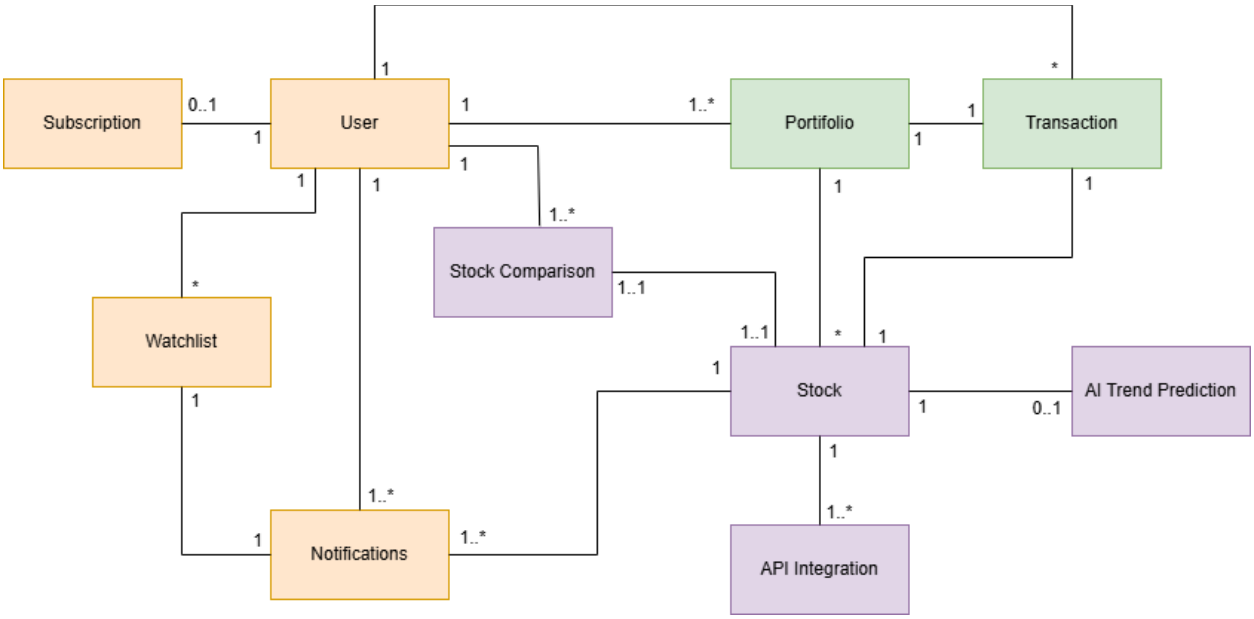
SockShark's architecture is built upon the core principles of Clean Architecture, which aims to create systems that are independent of frameworks, testable, and maintainable. This design follows the SOLID principles, making the system strong and easy to change.

- Single/Closed Principle (OCP): StockShark is designed to be open for extension but closed for modification. New features are added by introducing new code, rather than altering existing, stable code.

- Dependency Inversion Principle (DIP): High level modules, such as use cases that implement business logic, do not depend on low level modules, such as data access layers. Instead, low level modules depend on abstractions defined by high level modules.

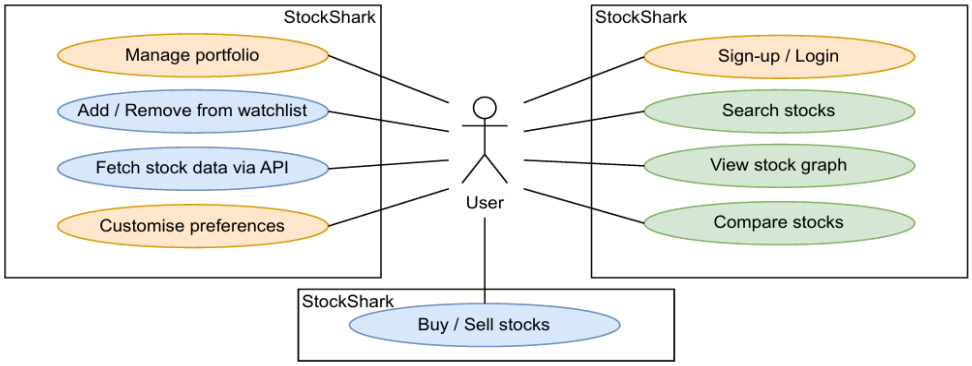
By separating these layers, we create a system that's flexible, maintainable, and easy to understand. This makes it easier for our team to add new features, fix bugs, and adapt to changing requirements, ultimately providing you with a better StockShark experience.

## Business Concept Model




The Business Concept Model is structured to provide a user-centric, scalable, and modular system for managing stock investments and transactions. At its core, the User entity connects to essential functions like portfolios, transactions, subscriptions, watchlists, and notifications, ensuring seamless interaction with the platform. The Stock entity is deeply integrated with real-time market updates via API integration, AI trend predictions, and stock comparisons, allowing for data-driven decision-making. The model also enforces flexibility, allowing users to manage multiple portfolios and transactions while ensuring that each action is properly tracked. The relational structure supports future expansion, such as adding new AI models, additional investment tools, or automation, without disrupting existing functionalities. By balancing functionality and scalability, this design enables a robust financial system that caters to both beginner and advanced investors.

# Use Case Model

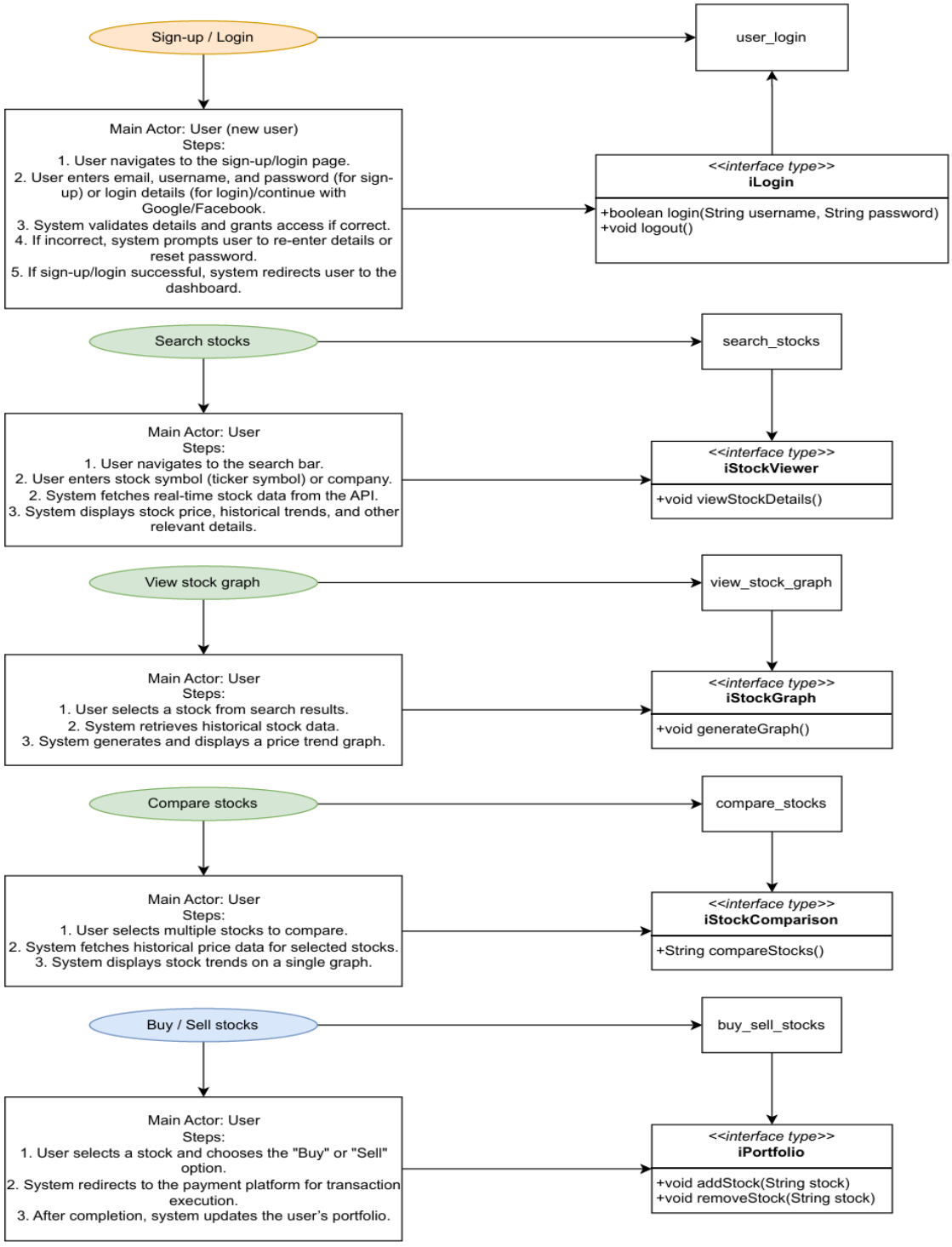


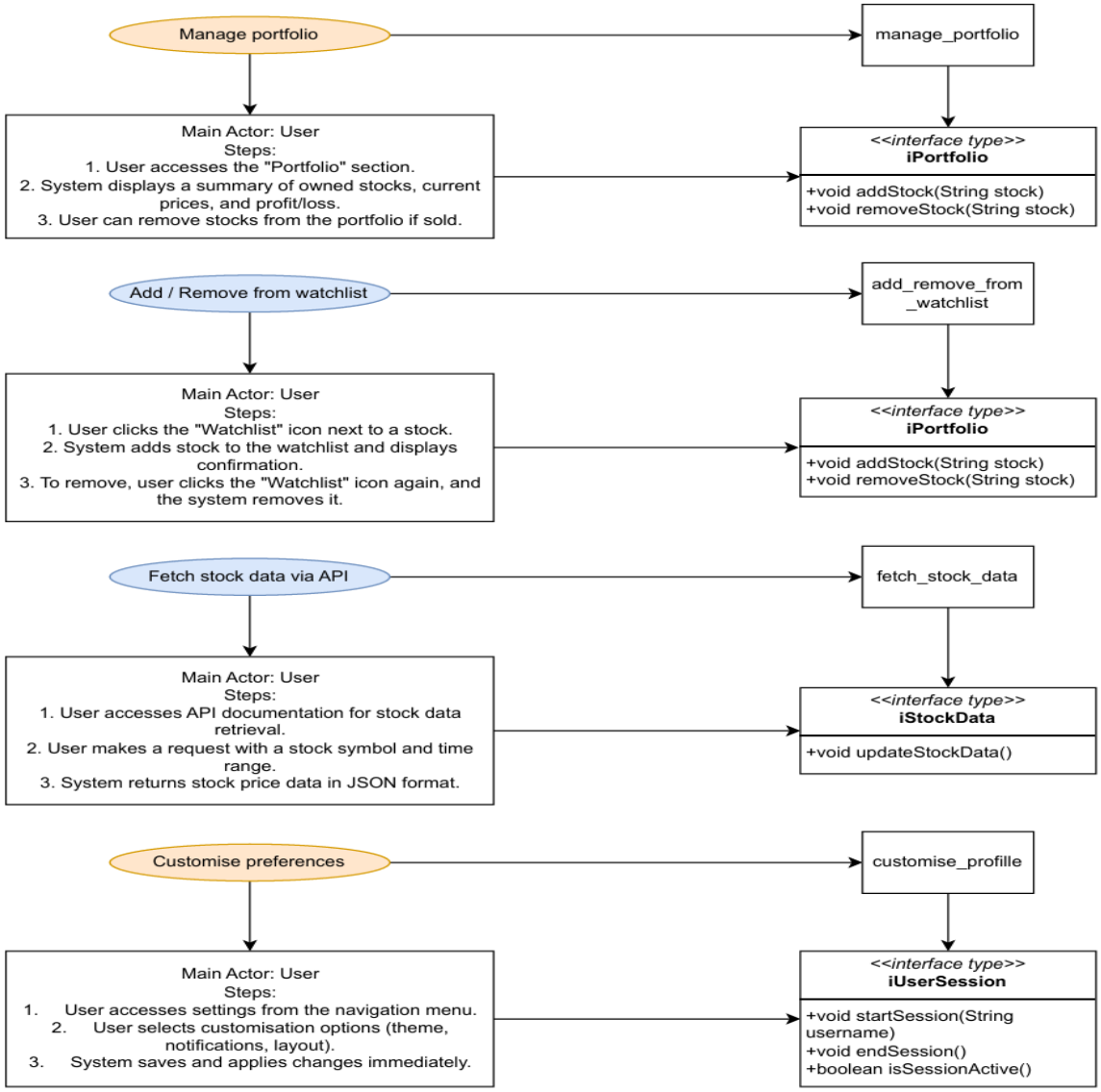

The StockShark Use Case Diagram is structured to provide a clear and user-centric experience, dividing functionalities into three main categories: portfolio management, stock exploration, and trading transactions.

Users can manage their investments by adding or removing stocks from their watchlist, customizing preferences, and fetching real-time stock data via API. The system also supports stock research through features like stock search, comparison, and historical graph visualization.

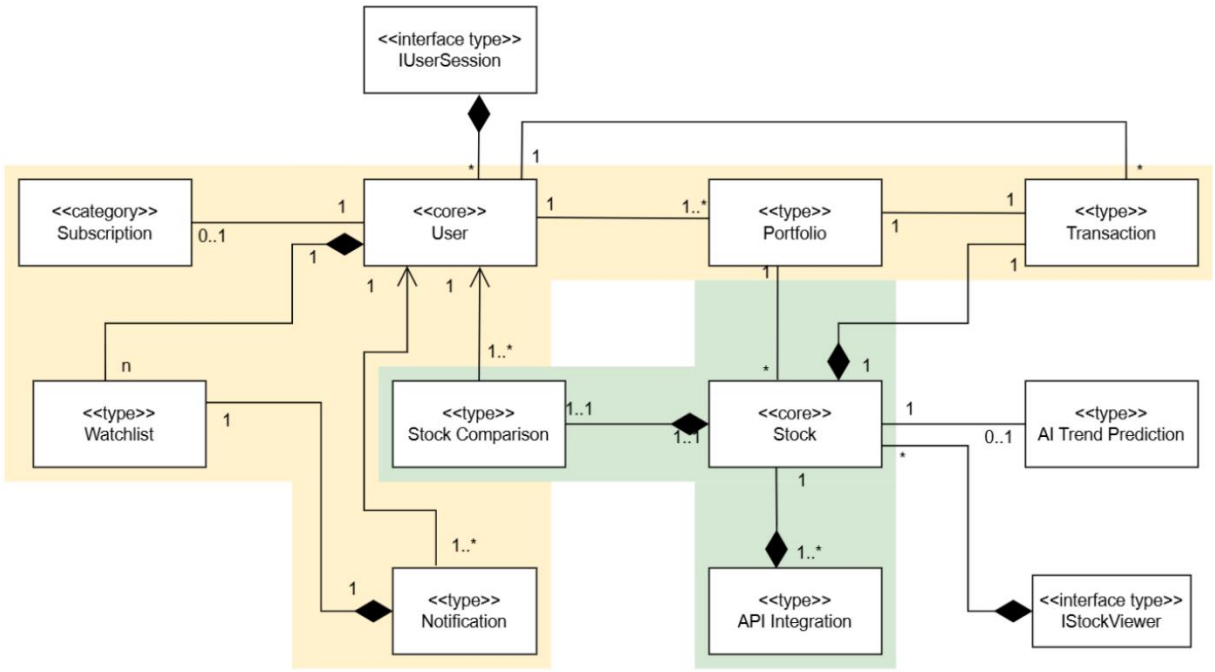
Additionally, the buy/sell stocks functionality ensures seamless trading operations. The inclusion of sign-up/login enhances security and access control. This structured approach ensures scalability, allowing for future enhancements like AI-driven insights or automated trading while maintaining an intuitive and efficient user workflow.

# System Interfaces



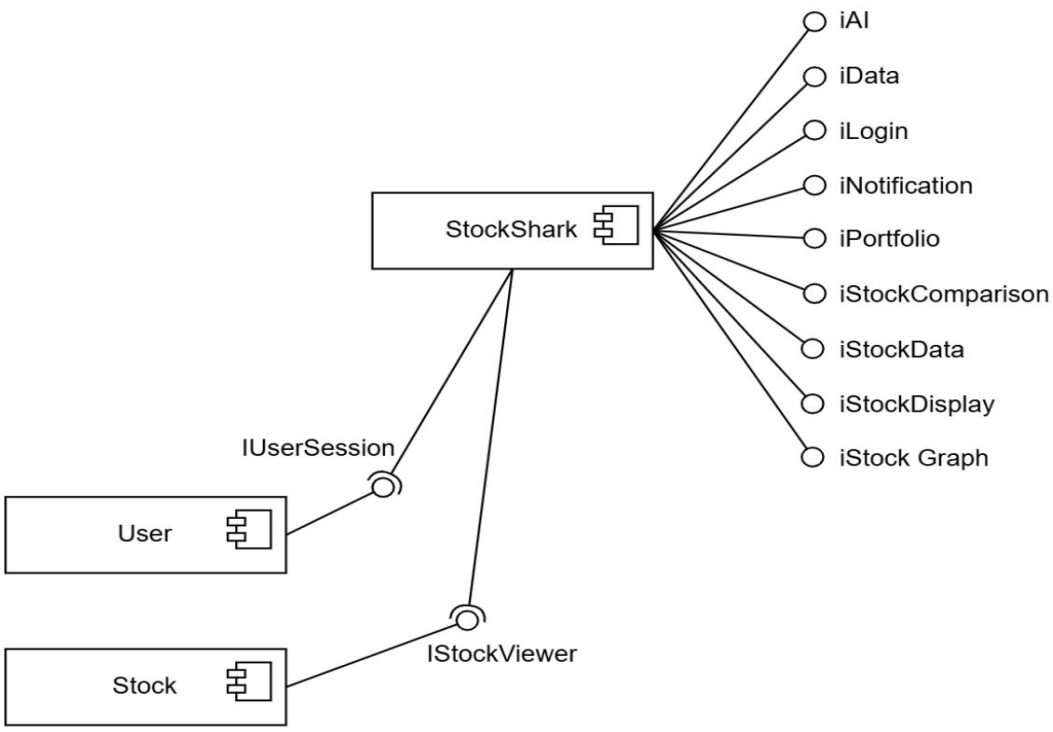

Manage portfolio
manage_portfolio
Main Actor: User
Steps:
1. User accesses the "Portfolio" section.
2. System displays a summary of owned stocks, current prices, and profit/loss.
3. User can remove stocks from the portfolio if sold.
<<interface type>>
iPortfolio
+void addStock(String stock)
+void removeStock(String stock)
Add / Remove from watchlist
add_remove_from_watchlist
Main Actor: User
Steps:
1. User clicks the "Watchlist" icon next to a stock.
2. System adds stock to the watchlist and displays confirmation.
3. To remove, user clicks the "Watchlist" icon again, and the system removes it.
<<interface type>>
iPortfolio
+void addStock(String stock)
+void removeStock(String stock)
Fetch stock data via API
fetch_stock_data
Main Actor: User
Steps:
1. User accesses API documentation for stock data retrieval.
2. User makes a request with a stock symbol and time range.
3. System returns stock price data in JSON format.
<<interface type>>
iStockData
+void updateStockData()
Customise preferences
customise_profille
Main Actor: User
Steps:
1. User accesses settings from the navigation menu.
2. User selects customisation options (theme, notifications, layout).
3. System saves and applies changes immediately.
<<interface type>>
iUserSession
+void startSession(String username)
+void endSession()
+boolean isSessionActive()

# Business Type Model




The Business Type Model in the diagram is structured to represent a scalable and modular stock trading system with distinct core entities, functional types, and interfaces. At the center, the User entity acts as the primary connection point, interacting with subscriptions, portfolios, transactions, notifications, and watchlists. The model ensures that users can manage their investments efficiently while maintaining real-time stock tracking through API integrations. The Stock entity, another core component, connects to stock comparison, AI trend prediction, and API integration, ensuring data-driven decision-making. Additionally, interface types like IUserSession and IStockViewer support session management and stock visualization, enhancing the platform's usability. The composition relationships (diamonds) indicate ownership structures, ensuring that key elements like portfolios, watchlists, and notifications are user-specific, while stocks exist independently. This model is designed for flexibility and expansion, allowing future enhancements such as automated trading, advanced analytics, or AI-driven insights without disrupting the core architecture.

# Initial System Architecture




The Initial System Architecture diagram represents the foundational structure of the StockShark platform, showcasing key system components and their interactions. At the core, StockShark serves as the central system, connecting various interfaces such as iAI, iData, iLogin, iNotification, iPortfolio, iStockComparison, iStockData, iStockDisplay, and iStockGraph, ensuring modularity and extendability. The User entity is linked through the IUserSession interface, allowing authentication and session management, ensuring secure access to the system. Meanwhile, the Stock entity interacts with StockShark via the IStockViewer interface, enabling real-time stock data retrieval and visualization. This architecture provides a clear separation of concerns, ensuring scalability, maintainability, and integration of advanced functionalities like AI-based trend analysis and stock comparison. The structured design allows for future enhancements, such as automated trading or additional analytics, while keeping the system flexible and user-friendly.

# UML2 Composite Component

The System Architecture Diagram for StockShark is structured to ensure scalability, modularity, and efficient data processing, dividing responsibilities across distinct layers. The Presentation Layer provides user interaction through notification systems, stock comparison dashboards, and graphical displays. The Business Layer handles core functionalities such as user management, stock comparison, and AI analysis, ensuring smooth execution of operations. Meanwhile, the Data Handling Layer integrates a database, external APIs, and data aggregation services to facilitate real-time stock data retrieval and processing. The component-based approach allows for seamless communication between modules, improving maintainability and future scalability. This design ensures a secure, data-driven, and user-friendly stock trading system, capable of integrating advanced analytics and AI-driven insights.

# Code-Related Deliverables

## Implementation of Clean Architecture Principles

Single Responsibility Principle (SRP) - each class has one responsibility.

Open Close Principle (OCP) - you can extend but not modify.

Liskov Substitution Principle (LSP) - subtypes are compatible with base class.

Interface Segregation Principle (ISP) - avoiding implementing unnecessary methods.

Dependency Inversion Principle (DIP) - have abstract methods instead of concrete coding implementations.

## Component and Interface Implementation

# Team Management

## Team Coordination:

Team Leader: Evan Balson – provided direction of Sprint 2 tasks, actively tracked progress and ensured completion. In charge of System Interfaces, composite component diagram and high-fidelity wire frames. Coding implementation: Database, AI analyzer and API Service.

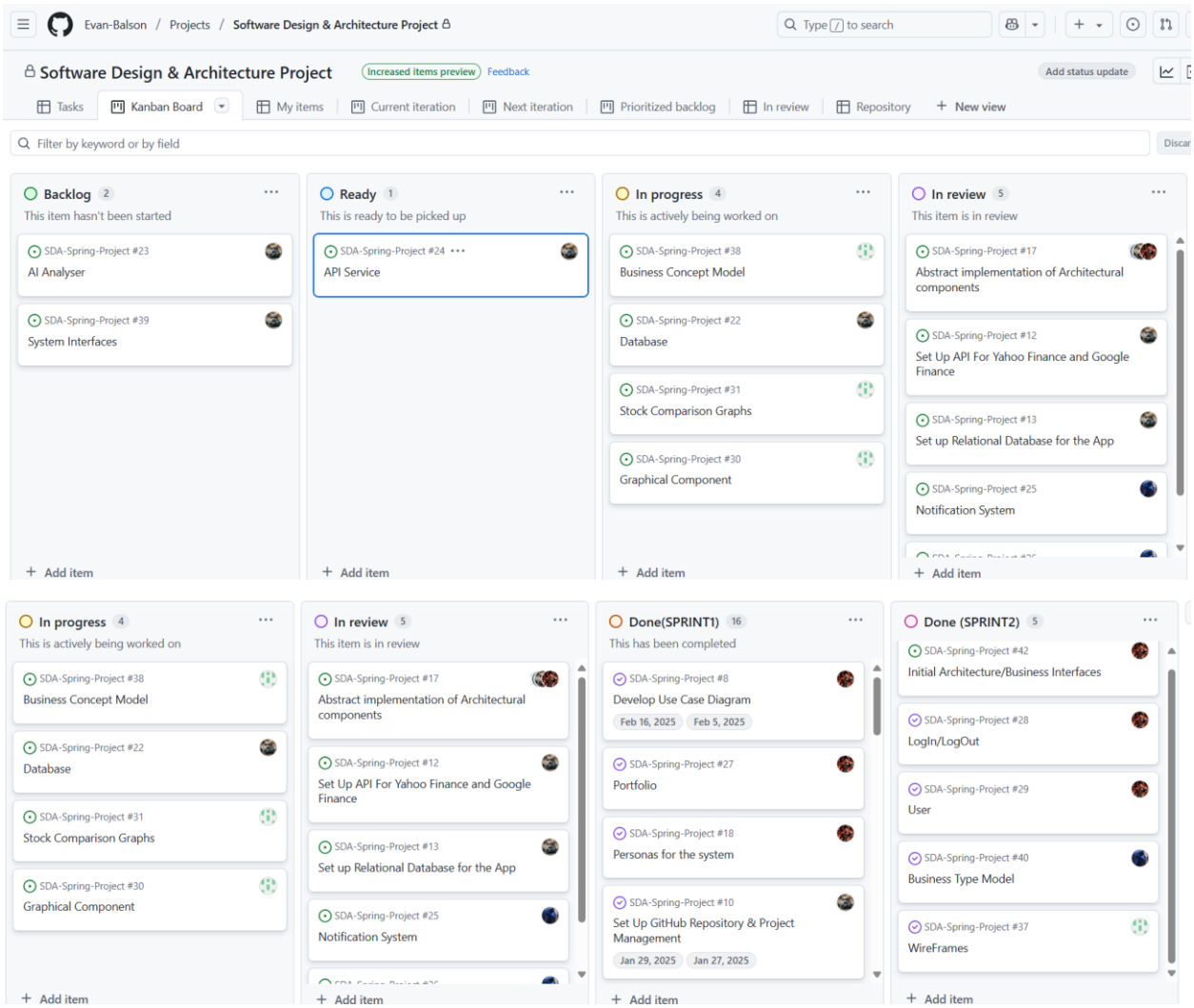
Scrum Master:  Maahia Rahman – scheduled weekly team meetings. In charge of Business Concept Model and low fidelity wireframes. Coding implementation: Graphical component and stock comparison graph.

Mahbouba: In charge of Business Type Model. Coding implementation: Notification system and stock comparison session.

Hala: In charge of Use Case, Initial Architecture and Business Interfaces. Coding implementation: Portfolio, Login/Logout and User.

Tasks were split equally amongst the group based off strengths and preference. To ensure the main branch was not disrupted, we used pull requests for double checking our implementations.

# GitHub Updates:



# References:

[1]. GeeksforGeeks, *“Complete Guide to Clean Architecture”, [Online], Available:* Complete Guide to Clean Architecture - GeeksforGeeks, [Accessed: March. 16, 2025]