Contents

Software Architecture and Design Project Report

Hand Crafted By:

Rodas Samson Gebrtensea

Haris Sabir

Rigers Bushi

Kaue Goncalves Ravagnani (Student)

Compound Components

User Class:

The User class encapsulates various components such as name, balance, transactionHistory, and ownedStocks.

It combines behaviors like depositing money, withdrawing money, purchasing stocks, selling stocks, and maintaining transaction history.

StockExchange Class:

The StockExchange class manages available stocks and stock transactions.

It uses a Map<String, Stock> to store available stocks and provides methods to buy and sell stocks.

Main Class:

The Main class serves as the main entry point of the application.

It orchestrates interactions between the User and StockExchange to simulate a user interacting with a stock market system.

Stock Class:

Although not explicitly mentioned as a compound component, the Stock class represents individual stocks and their properties, which is used within the StockExchange.

In terms of composition and interaction:

The Main class orchestrates interactions between a User and a StockExchange, creating a higher-level system.

The User class encapsulates behavior related to user actions such as managing balance, stocks, and transaction history.

The StockExchange class manages available stocks and stock transactions, providing methods for buying and selling stocks.

These components work together to simulate a basic stock market system where users can interact with available stocks, make purchases, and view transaction history. The composition and interaction between these components demonstrate the compound nature of the system, where smaller components are combined to create a larger, more complex application.

Architectural Patterns

Model-View-Controller (MVC) Pattern:

The program's structure resembles the MVC pattern:

Model: The User, Stock, and StockExchange classes represent the data model of the application, encapsulating the business logic and data operations.

View: The interaction with the user through the console-based menu system (Main class) can be considered as the view layer.

Controller: The Main class acts as the controller, handling user input, invoking appropriate business logic methods (from User and StockExchange), and updating the view accordingly.

Facade Pattern:

The Main class serves as a facade that simplifies the interface for interacting with the underlying complex subsystems (User and StockExchange). It presents a unified interface (handleWallet and handleStockMarket) to perform user operations.

Repository Pattern (Data Mapper Pattern):

Although not explicitly implemented in the provided code, the concept of data access and persistence (in-memory storage of user data, transaction history, and available stocks) aligns with the repository pattern. The User and StockExchange classes manage data related to users and stocks, providing methods to interact with and manipulate this data.

Command Pattern:

The menu-driven user interface in the Main class can be viewed as employing a command pattern, where user choices (commands) are encapsulated into objects (handleWallet and handleStockMarket methods) that can be executed based on user input.

Singleton Pattern (for Scanner):

Although not a core architectural pattern, the use of a single Scanner instance throughout the program (created in Main) demonstrates a singleton-like behavior, ensuring that there is a single point of access for user input.

Dependency Injection (DI):

The Main class injects instances of User and StockExchange into the methods (handleWallet and handleStockMarket), allowing for loose coupling between the main logic and the business logic classes.

These architectural patterns and design principles contribute to a well-structured and modular design, promoting separation of concerns, flexibility, and maintainability of the

overall system. The specific patterns highlighted showcase common strategies used in software design to address scalability, extensibility, and readability

Team Meetings Recorded Times

Tuesday, March 27th

Time: 10:00 AM

Duration: 1 hour

Thursday, April 5th

Time: 2:00 PM

Duration: 1.5 hours

Monday, April 16th

Time: 9:00 AM

Duration: 2 hours

Wednesday, April 24th

Time: 3:00 PM

Duration: 1 hour

Test Cases

Test Case Table: User Menu Interaction

Test Case ID	TC-003
Objective	To verify that the user menu handles selections properly, displaying respective actions or error messages based on user input.
Test Description	This test case will verify each menu option (View Wallet, Enter Stock Market, and Exit), as well as the response to an invalid option.
Pre-Conditions	User must be logged in and the application started with necessary variables (scanner, user, exchange) initialized.

Test Steps	Test Data	Expected Result
1. Observe initial greetings and options displayed.	-	"Welcome, [User's Name]!" followed by menu options "1. View Wallet", "2. Enter Stock Market", "3. Exit" are displayed.
2. Input option 1 for "View Wallet".	1	Wallet interface or operations are initiated (handled by handleWallet).
3. Input option 2 for "Enter Stock Market".	2	Stock market interface or operations are initiated (handled by handleStockMarket).
4. Input an invalid option 5.	5	"Invalid option. Please try again." message is displayed.
5. Input option 3 to exit the application.	3	"Exiting..." message is displayed, and the program terminates successfully.
Post-Conditions	-	Application must close properly, and system resources released (e.g., scanner closed).
Actual Results	To be filled after test execution	-
Status	To be determined	-
Created By	[Rodas Samson Gebrtensea]	
Test Environment	Java Runtime Environment on [Operating System], Console-based execution	

Test Steps	Test Data	Expected Result
Comments	Ensure all resources are freed and application exits cleanly to avoid memory leaks or resource locking.	

Test Case Table: Verify Stock Purchase Functionality

Field	Details
Test Case ID	TC-004
Test Objective	To verify the functionality of the "Buy stock" option, ensuring the user can select a stock, enter the desired purchase quantity, handle invalid inputs correctly, and utilize an exit option if needed.
Test Description	This test case aims to test the purchase process of stocks, including correct and incorrect symbol inputs, valid and invalid quantity inputs, and the ability to exit the purchase prompt.
Pre-Conditions	- User is logged in. - The application is running, with scanner, user, and exchange initialized. - exchange has methods displayAvailableStocks() and getStock(String symbol). - user has a method purchaseStock(Stock stock, int quantity).
Test Steps	Test Data
1. Choose to buy stock.	Enter 1 at main menu.
2. Enter a valid stock symbol.	"AAPL"
3. Enter a valid quantity.	100
4. Attempt to buy with an invalid symbol.	"XYZ"
5. Exit purchase option.	"B"
Post-Conditions	- For valid purchases: User's stock holdings updated. - For invalid/exit: No change in user's stock holdings.
Actual Results	To be filled during test execution.
Status	To be determined post-execution.
Created By	Rigers Bushi
Test Environment	Java Runtime Environment on [Operating System], Console-based execution
Comments	Include checks for handling non-numeric input for quantity and test boundary conditions for quantity (e.g., zero, negative values). Ensure error messages are

Field	Details
	appropriately displayed and that the system correctly handles exit scenario without processing further inputs.

Test Case Table: Bank Account Operations

Test Case for Deposit Method

Field	Details
Test Case ID	TC-005
Test Objective	To verify that the deposit method correctly handles valid and invalid deposit amounts.
Test Description	This test case ensures the deposit method accepts only positive values and rejects non-positive values with an appropriate error message.
Pre-Conditions	Bank account is initialized with a starting balance (e.g., $1000).
Test Steps	Test Data
1. Deposit a valid positive amount.	$200
2. Attempt to deposit zero.	$0
3. Attempt to deposit a negative amount.	-$50
Post-Conditions	Balance must reflect only valid transactions.
Actual Results	To be filled during test execution.
Status	To be determined post-execution.
Created By	[Your Name]
Test Environment	Java Runtime Environment on [Operating System], IDE or console-based execution
Comments	None

Test Case for Withdraw Method

Field	Details
Test Case ID	TC-006
Test Objective	To verify that the withdraw method correctly handles withdrawals with respect to the available balance and rejects inappropriate withdrawal amounts.
Test Description	This test case checks the withdrawal method for various scenarios including valid withdrawals, overdrawing, and non-positive values.
Pre-Conditions	Bank account is initialized with a starting balance (e.g., $1000).
Test Steps	Test Data
1. Withdraw a valid amount within the balance limit.	$200
2. Attempt to withdraw more than the current balance.	$1500
3. Attempt to withdraw a negative amount.	-$100
4. Attempt to withdraw zero.	$0
Post-Conditions	Balance must reflect only valid transactions.
Actual Results	To be filled during test execution.
Status	To be determined post-execution.
Created By	[Your Name]
Test Environment	Java Runtime Environment on [Operating System], IDE or console-based execution
Comments	Consider including additional checks for decimal values if the system should handle fractional amounts.

Code Snippets

```
        while (true) {
            System.out.println("Welcome, " + user.getName() + "!");
            System.out.println("Choose an option:");
            System.out.println("1. View Wallet");
            System.out.println("2. Enter Stock Market");
            System.out.println("3. Exit");

            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    // Wallet operations
                    handleWallet(user, scanner);
                    break;
                case 2:
                    // Stock market operations
                    handleStockMarket(user, exchange, scanner);
                    break;
                case 3:
                    System.out.println("Exiting...");
                    scanner.close();
                    System.exit(0);
                    break;
                default:
                    System.out.println("Invalid option. Please try again.");
            }
```

```
 switch (choice) {
                case 1:
                    // Buy stock
                    System.out.println("Available Stocks:");
                    exchange.displayAvailableStocks();
                    System.out.println("Enter stock symbol to buy (or 'B' to
exit):");
                    String buySymbol = scanner.next();
                    if (buySymbol.equalsIgnoreCase("B")) {
                        break;
                    }
                    System.out.print("Enter quantity to buy: ");
                    int quantityToBuy = scanner.nextInt();
                    Stock stockToBuy = exchange.getStock(buySymbol);
                    if (stockToBuy == null) {
                        System.out.println("Invalid stock symbol.");
                        break;
                    }
                    System.out.println(user.purchaseStock(stockToBuy,
quantityToBuy));
                    break;
```

```java
    public String deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
            addToHistory("Deposit: $" + amount);
            return "Successfully deposited $" + amount;
        } else {
            return "Invalid amount. Please enter a positive number.";
        }
    }

    public String withdraw(double amount) {
        if (amount > 0 && amount <= this.balance) {
            this.balance -= amount;
            addToHistory("Withdrawal: $" + amount);
            return "Successfully withdrew $" + amount;
        } else if (amount <= 0) {
            return "Invalid amount. Please enter a positive number.";
        } else {
            return "Insufficient funds.";
        }
    }
```