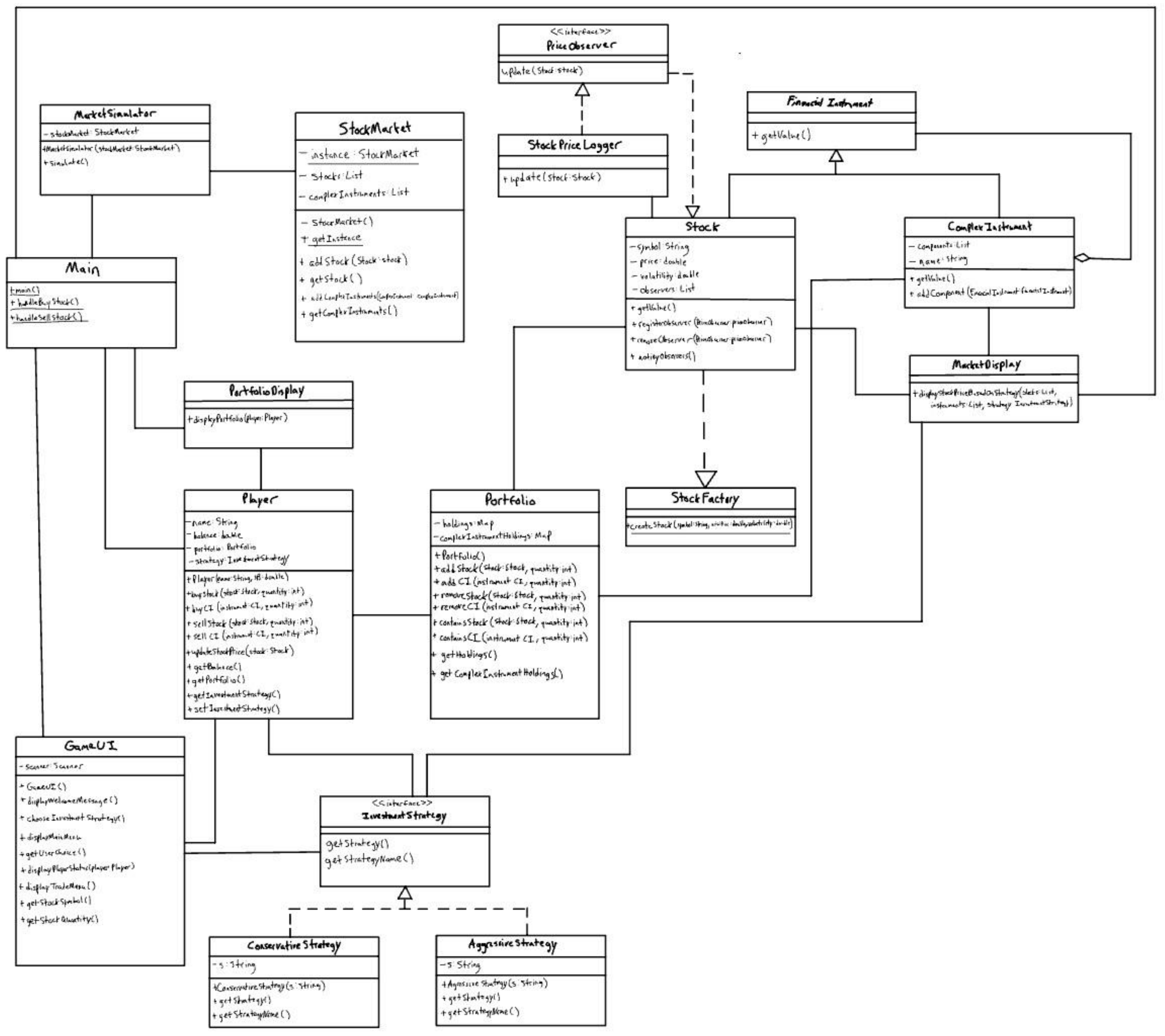
Stock Market Simulation Game



<<interface>>
PriceObserver
update(Stock stock)
StockPriceLogger
+update(stock: Stock)
Financial Instrument
+ getValue()
MarketSimulator
- stockMarket: StockMarket
+MarketSimulator(stockMarket: StockMarket)
+simulate()
StockMarket
- instance: StockMarket
- stocks: List
- complexInstruments: List
- StockMarket()
+ getInstance
+ addStock(Stock: stock)
+ getStock()
+ addComplexInstruments(ComplexInstrument: complexInstrument)
+ getComplexInstruments()
Stock
- symbol: String
- price: double
- volatility: double
- observers: List
+ getValue()
+ registerObserver(PriceObserver priceObserver)
+ removeObserver(PriceObserver priceObserver)
+ notifyObservers()
ComplexInstrument
- components: List
- name: String
+ getValue()
+ addComponent(FinancialInstrument financialInstrument)
Main
+main()
+ handleBuyStock()
+handleSellStock()
MarketDisplay
+displayStockPriceBasedOnStrategy(stocks: List, instruments: List, strategy: InvestmentStrategy)
PortfolioDisplay
+displayPortfolio(player: Player)
Player
-name: String
- balance: double
- portfolio: Portfolio
- strategy: InvestmentStrategy
+Player(name: String, IB: double)
+buyStock(stock: Stock, quantity: int)
+buyCI(instrument: CI, quantity: int)
+sellStock(stock: Stock, quantity: int)
+sellCI(instrument: CI, quantity: int)
+updateStockPrice(stock: Stock)
+ getBalance()
+getPortfolio()
+getInvestmentStrategy()
+setInvestmentStrategy()
Portfolio
- holdings: Map
- complexInstrumentHoldings: Map
+Portfolio()
+addStock(stock: Stock, quantity: int)
+addCI(instrument CI, quantity: int)
+removeStock(stock: Stock, quantity: int)
+removeCI(instrument CI, quantity: int)
+containsStock(stock: Stock, quantity: int)
+containsCI(instrument CI, quantity: int)
+ getHoldings()
+ getComplexInstrumentHoldings()
StockFactory
+createStock(symbol: String, price: double, volatility: double)
GameUI
- Scanner: Scanner
+ GameUI()
+ displayWelcomeMessage()
+ chooseInvestmentStrategy()
+ displayMainMenu
+getUserChoice()
+displayPlayerStatus(player Player)
+displayTradeMenu()
+getStockSymbol()
+getStockQuantity()
<<interface>>
InvestmentStrategy
getStrategy()
getStrategyName()
ConservativeStrategy
- s: String
+ConservativeStrategy(s: String)
+getStrategy()
+getStrategyName()
AggressiveStrategy
-s: String
+AggressiveStrategy(s: String)
+getStrategy()
+getStrategyName()

About the Project:
The project is a stock market simulation game designed to provide an interactive and educational experience around stock market dynamics. In this simulation, players can buy and sell stocks and complex financial instruments, choose investment strategies, and observe the impact of market changes on their portfolio. The game includes a user interface for interacting with the market, viewing portfolio values, and making trading decisions. The simulation's backend handles market fluctuations, stock price updates, and portfolio management, offering a realistic experience of stock market trading.

How to Run:
Type "java Main" into the terminal to start the simulation. You will first be presented with the choice to choose your preferred investment strategy. This decision will decide how the stocks are sorted when you view the market. At this point, you can choose to view the market, your current portfolio, trade, or exit the game by typing in the corresponding number. If you choose to trade, you will be able to buy/sell stocks or buy/sell complex instruments (which are simply combinations of stocks in the market put into one fund).

Patterns:

Singleton:
- Purpose: Ensures that there is only one instance of the StockMarket class throughout the game. This is important because the stock market in the simulation needs to be a single, globally accessible entity that maintains consistent state across the entire simulation.
- Why Chosen: Singleton provides a global point of access to the StockMarket instance, which ensures data consistency and prevents issues that could arise from having multiple instances of the market, such as conflicting stock prices or investment strategies.

Composite:
- Purpose: Allows individual stocks and composite financial instruments (like ETFs) to be treated uniformly. A composite complex instrument can hold multiple stocks or other composites, enabling hierarchical portfolios.
- Why Chosen: This pattern provides flexibility in portfolio construction and simplifies operations on portfolios, as both simple and complex financial instruments can be managed using the same interface.

Factory:
- Purpose: Encapsulates the creation of Stock objects, allowing for controlled instantiation and potential future expansion in stock creation logic (like adding validation or complex initialization steps).
- Why Chosen: It abstracts the complexity of object creation from the client code, providing a simple interface for creating stocks while keeping the instantiation logic centralized and maintainable.

Strategy:

- Purpose: Enables selection of investment strategies. Players can choose between different strategies affecting how the market is listed, either in terms of least to most volatile (preferable for conservative investors) or most to least volatile (preferable for aggressive investors).
- Why Chosen: This pattern allows for the easy extension of investment strategies without altering the player or the market's core functionality, providing flexibility in strategy implementation.

<u>Observer:</u>

- Purpose: Used for notifying changes in stock prices when stocks are bought/sold.
- Why Chosen: It decouples the stock objects from their observers, allowing for easy addition or removal of different observers without modifying the stock's code. This pattern enhances adaptability in handling notifications.