

AI-Powered Trading Bot Documentation

1. Introduction

Project Description

AI-Powered Trading Bot is an automation system designed for trading in financial markets. The bot analyzes market data using various trading strategies and makes buy, sell, or hold decisions based on the analyzed data. This project leverages four key software design patterns: Template, Strategy, Command, and Observer to ensure flexibility, scalability, and maintainability.

Project Goals

The primary objective of this project is to develop an automated trading bot that can analyze market data and execute trading decisions based on predefined strategies. The bot supports multiple trading strategies, commands, and market data updates.

Technologies Used

The project is developed using Python, with the following key technologies:

- Python programming language
- yfinance library for market data
- Tkinter for user interface
- Matplotlib for real-time chart plotting.

2. System Limitations and Future Work

While the AI-Powered Trading Bot provides a comprehensive solution for trading automation, some limitations exist:

- The system's ability to handle large volumes of real-time data may require further optimization for higher frequency trading environments.
- The trading strategies currently implemented are relatively basic and may not be suitable for more complex market conditions.
- Error handling for trade execution, such as in cases of failure or market anomalies, needs further refinement.

Future work could focus on:

- Enhancing the scalability of the bot for larger datasets and real-time processing, potentially with a distributed architecture.
- Integrating more advanced trading strategies, such as machine learning models for prediction and decision-making.
- Adding features for improved error handling and transaction rollback mechanisms.
- Implementing a user interface for real-time monitoring and control of the trading bot.

3. Project Structure

Main Classes

1. **MarketDataProvider**: Fetches market data and notifies its observers about updates.
2. **TradingBot**: Implements the selected trading strategy and executes trading commands.
3. **TradingInvoker**: Invokes the commands such as buy, sell, or hold.

4. **AbstractTradingStrategy and its subclasses:** Defines the trading strategy logic.
5. **BuyCommand, SellCommand, HoldCommand:** Encapsulate the trading actions.

4. Code Review and Design Pattern Implementation

4.1 Template and Strategy Pattern

Both the **Template Pattern** and **Strategy Pattern** are employed to create a flexible trading system where various algorithms can be interchanged and executed without modifying the core structure of the bot.

Template Pattern:

The **Template Pattern** is used to define the skeleton of an algorithm in a base class, allowing the subclasses to implement specific steps of the algorithm. In this project, the `AbstractTradingStrategy` class acts as the template. It provides a `decide()` method that follows the algorithm steps: first, it validates the market data and then delegates the strategy-specific logic to the `execute_strategy()` method. This allows different trading strategies to be applied while maintaining a consistent process.

The template method is crucial for maintaining consistency in how trading decisions are made, regardless of which strategy is applied.

Strategy Pattern:

The **Strategy Pattern** is used to define a family of algorithms (in this case, trading strategies) and encapsulate them, making them interchangeable. In this project, different concrete strategies (such as Moving Average, Bollinger Bands, and Momentum) are implemented as separate classes, all inheriting from `AbstractTradingStrategy`.

The Strategy Pattern allows the `TradingBot` to dynamically select which strategy to use based on market conditions or user input. Each strategy implements its own version of the `execute_strategy()` method, which defines how the bot makes a decision (buy, sell, or hold) based on the market data.

By using both the Template and Strategy patterns, we ensure that:

1. The trading decision process remains the same (validated by the template method).
2. The specific algorithm or strategy used can be easily replaced without changing the core logic.

```
AbstractTradingStrategy.py X
AbstractTradingStrategy.py > AbstractTradingStrategy > execute_strategy
1 from abc import ABC, abstractmethod
2
3 class AbstractTradingStrategy(ABC):
4     def decide(self, market_data):
5         if not self.is_valid_data(market_data):
6             return "HOLD"
7         return self.execute_strategy(market_data)
8
9     def _is_valid_data(self, market_data):
10        return market_data and len(market_data) > 1
11
12    @abstractmethod
13    def execute_strategy(self, market_data):
14        pass
```

```
BollingerBandsStrategy.py X
BollingerBandsStrategy.py > BollingerBandsStrategy > calculate_standard_deviation
1 from AbstractTradingStrategy import AbstractTradingStrategy
2 import numpy as np
3
4 class BollingerBandsStrategy(AbstractTradingStrategy):
5     def execute_strategy(self, market_data):
6         mean = self.calculate_mean(market_data)
7         std_dev = self.calculate_standard_deviation(market_data, mean)
8
9         upper_band = mean + 2 * std_dev
10        lower_band = mean - 2 * std_dev
11
12        last_price = market_data[-1]
13
14        if last_price > upper_band:
15            return "SELL" # Overbought
16        elif last_price < lower_band:
17            return "BUY" # Oversold
18        else:
19            return "HOLD"
20
21    def calculate_mean(self, data):
22        return np.mean(data)
23
24    def calculate_standard_deviation(self, data, mean):
25        return np.std(data)
```

```
MomentumStrategy.py X
MomentumStrategy.py > MomentumStrategy > execute_strategy
1 from AbstractTradingStrategy import AbstractTradingStrategy
2
3 class MomentumStrategy(AbstractTradingStrategy):
4     def execute_strategy(self, market_data):
5         if market_data[-1] > market_data[-2]:
6             return "BUY"
7         elif market_data[-1] < market_data[-2]:
8             return "SELL"
9         else:
10            return "HOLD"
```

```
MovingAverageStrategy.py X
MovingAverageStrategy.py > MovingAverageStrategy > calculate_moving_average
1 from AbstractTradingStrategy import AbstractTradingStrategy
2
3 class MovingAverageStrategy(AbstractTradingStrategy):
4     def __init__(self, short_window=3, long_window=5):
5         self.short_window = short_window
6         self.long_window = long_window
7
8     def execute_strategy(self, market_data):
9         if len(market_data) < self.long_window:
10            return "HOLD"
11
12         short_ma = self.calculate_moving_average(market_data, self.short_window)
13         long_ma = self.calculate_moving_average(market_data, self.long_window)
14
15         if short_ma > long_ma:
16             return "BUY"
17         elif short_ma < long_ma:
18             return "SELL"
19         else:
20             return "HOLD"
21
22     def calculate_moving_average(self, data, window):
23         return sum(data[-window:][0]) / window
```

```
RsiStrategy.py X
RsiStrategy.py > RsiStrategy > _calculate_rsi
1 from AbstractTradingStrategy import AbstractTradingStrategy
2
3 class RsiStrategy(AbstractTradingStrategy):
4     def execute_strategy(self, market_data):
5         rsi = self._calculate_rsi(market_data)
6         if rsi < 30:
7             return "BUY"
8         elif rsi > 70:
9             return "SELL"
10        else:
11            return "HOLD"
12
13    def _calculate_rsi(self, market_data):
14        gains = losses = 0
15        for i in range(1, len(market_data)):
16            diff = market_data[i][0] - market_data[i - 1][0]
17            if diff > 0:
18                gains += diff
19            else:
20                losses -= diff
21        if losses == 0:
22            return 100
23        rs = gains / losses
24        return 100 - (100 / (1 + rs))
```

```
TradingStrategy.py X
TradingStrategy.py > ...
1 from AbstractTradingStrategy import AbstractTradingStrategy
2 class TradingStrategy:
3     def __init__(self):
4         self._strategies = {}
5         self._current_strategy = None
6
7     def register_strategy(self, name, strategy: AbstractTradingStrategy):
8         if not isinstance(strategy, AbstractTradingStrategy):
9             raise TypeError("Strategy must inherit from AbstractTradingStrategy.")
10        self._strategies[name] = strategy
11
12    def set_strategy(self, name):
13        if name not in self._strategies:
14            raise ValueError(f"Strategy {name} is not registered.")
15        self._current_strategy = self._strategies[name]
16
17    def get_current_strategy(self) -> AbstractTradingStrategy:
18        if self._current_strategy is None:
19            raise ValueError("No strategy is currently set.")
20        return self._current_strategy
21
22    def get_strategy_names(self):
23        return list(self._strategies.keys())
24
25
```

Explanation:

- **Template Method (decide):** The decide method is the template method that executes the general steps of validating market data and executing the strategy.

- **Concrete Strategy Classes:** Each concrete strategy (like MomentumStrategy, BollingerBandsStrategy, etc.) implements the `execute_strategy()` method, which contains specific logic for deciding whether to buy, sell, or hold.

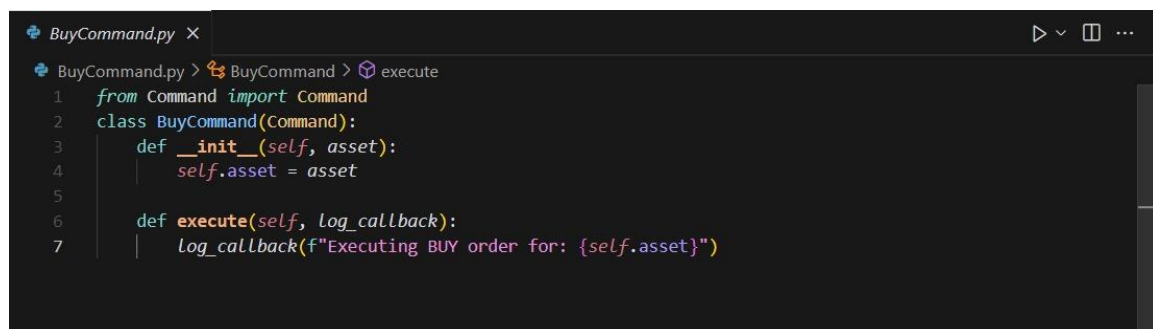
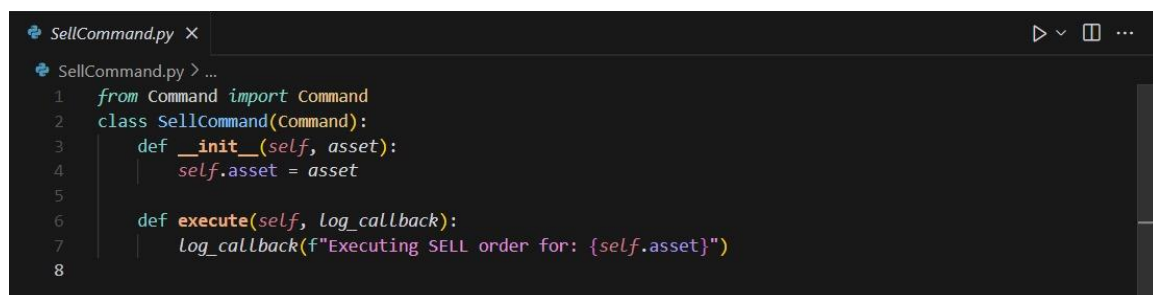
This combination of the **Template Pattern** and the **Strategy Pattern** provides flexibility and allows for easy swapping between strategies while maintaining a consistent decision-making process.

4.2 Command Pattern

The Command Pattern is used to encapsulate trading actions (buy, sell, hold) into objects, making it easier to execute and manage them. This allows actions to be passed as parameters to invokers, decoupling the command from the objects that execute it.

Code Implementation

- The Command class is the abstract base class for all command objects, defining the `execute` method.
- BuyCommand, SellCommand, and HoldCommand extend this class, encapsulating the logic for executing each type of command.
- The TradingInvoker class is responsible for executing the commands.

A screenshot of a code editor showing the implementation of the BuyCommand class. The file is named 'BuyCommand.py'. The code imports the 'Command' class from a module named 'Command'. It then defines a 'BuyCommand' class that inherits from 'Command'. The class has an '__init__' method that takes 'self' and 'asset' as arguments and assigns 'asset' to 'self.asset'. It also has an 'execute' method that takes 'self' and 'log_callback' as arguments and calls 'log_callback' with the string 'Executing BUY order for: {self.asset}'.A screenshot of a code editor showing the implementation of the SellCommand class. The file is named 'SellCommand.py'. The code imports the 'Command' class from a module named 'Command'. It then defines a 'SellCommand' class that inherits from 'Command'. The class has an '__init__' method that takes 'self' and 'asset' as arguments and assigns 'asset' to 'self.asset'. It also has an 'execute' method that takes 'self' and 'log_callback' as arguments and calls 'log_callback' with the string 'Executing SELL order for: {self.asset}'.



```
HoldCommand.py X
HoldCommand.py > ...
1 from Command import Command
2 class HoldCommand(Command):
3     def execute(self, log_callback):
4         log_callback("No action taken. HOLD.")
5
```

Explanation of the Code

- **Command Interface:** The Command class defines the execute method, ensuring that all concrete command classes implement it.
- **Concrete Commands:** BuyCommand, SellCommand, and HoldCommand encapsulate specific trading actions. The TradingInvoker class takes these commands and executes them when needed.

4.3 Observer Pattern

The Observer Pattern allows objects (observers) to subscribe to updates from a subject. In this project, MarketDataProvider is the subject that notifies the trading bot whenever there is an update in market data.

Code Implementation

- MarketDataProvider implements the subject role and manages a list of observers.
- MarketDataObserver is the abstract class that must be implemented by the observers, such as the TradingBot, to handle market data updates.


```
MarketDataProvider.py X
MarketDataProvider.py > MarketDataProvider > _init_
1  import yfinance as yf
2  import time
3  from pandas import DataFrame
4  from MarketDataSubject import MarketDataSubject
5
6  class MarketDataProvider(MarketDataSubject):
7      def __init__(self, symbol, interval=5, period="1d", data_interval="1m", log_callback=None):
8          self.symbol = symbol
9          self.interval = interval
10         self.period = period
11         self.data_interval = data_interval
12         self.market_data: DataFrame | None = None
13         self.observers = []
14         self.running = False
15         self.log_callback = log_callback
16
17     def log(self, message):
18         if self.log_callback:
19             self.log_callback(message)
20         else:
21             print(message)
22
23     def add_observer(self, observer):
24         self.observers.append(observer)
25
26     def remove_observer(self, observer):
27         self.observers.remove(observer)
28
29     def notify_observers(self, market_data):
30         for observer in self.observers:
31             observer.on_market_data_update(market_data)
32
```

```

33     def fetch_asset_data(self):
34         try:
35             self.log(f"Fetching data for {self.symbol}...")
36             data = yf.download(tickers=self.symbol, period=self.period, interval=self.data_interval)
37             if data.empty:
38                 self.log(f"No data found for {self.symbol}.")
39                 return None
40             self.log(f"Data fetched successfully for {self.symbol}.")
41             return data
42         except Exception as e:
43             self.log(f"Error fetching data for {self.symbol}: {e}")
44             return None
45
46     def load_market_data(self):
47         self.market_data = self.fetch_asset_data()
48         if self.market_data is not None:
49             self.log(f"Loaded market data for {self.symbol}.")
50         else:
51             self.log(f"Failed to load market data for {self.symbol}.")
52
53     def start(self):
54         self.running = True
55         if self.market_data is None or self.market_data.empty:
56             self.log("No market data to process.")
57             return
58         for index in range(len(self.market_data)):
59             if not self.running:
60                 break
61             row = self.market_data.iloc[index]
62             price_history = self.market_data['Close'].iloc[:index + 1].values.tolist()
63             self.log(f"Simulated Update: {row.name} - Close: {row['Close']}")
64             self.notify_observers(price_history)
65
66             time.sleep(self.interval)

```

```

67
68     def stop(self):
69         self.running = False
70         if self.log_callback:
71             self.log("Market data streaming stopped.")

```

MarketDataSubject.py X

MarketDataSubject.py > ...

```

1  from abc import ABC, abstractmethod
2
3  class MarketDataSubject(ABC):
4      @abstractmethod
5      def add_observer(self, observer):
6          pass
7      @abstractmethod
8      def remove_observer(self, observer):
9          pass
10     @abstractmethod
11     def notify_observers(self, market_data):
12         pass

```

MarketDataObserver.py X

MarketDataObserver.py > MarketDataObserver > on_market_data_update

```

1  from abc import ABC, abstractmethod
2  class MarketDataObserver(ABC):
3      @abstractmethod
4      def on_market_data_update(self, market_data):
5          pass

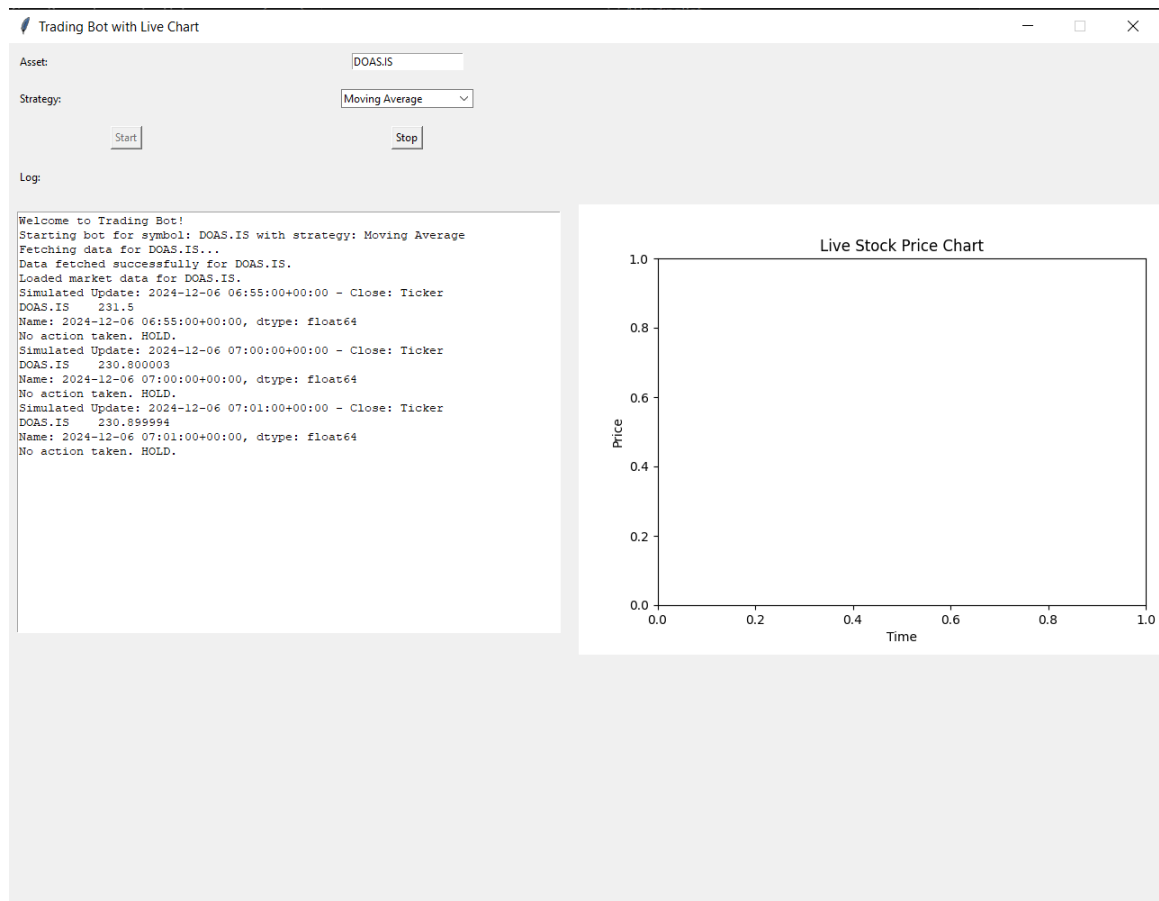
```

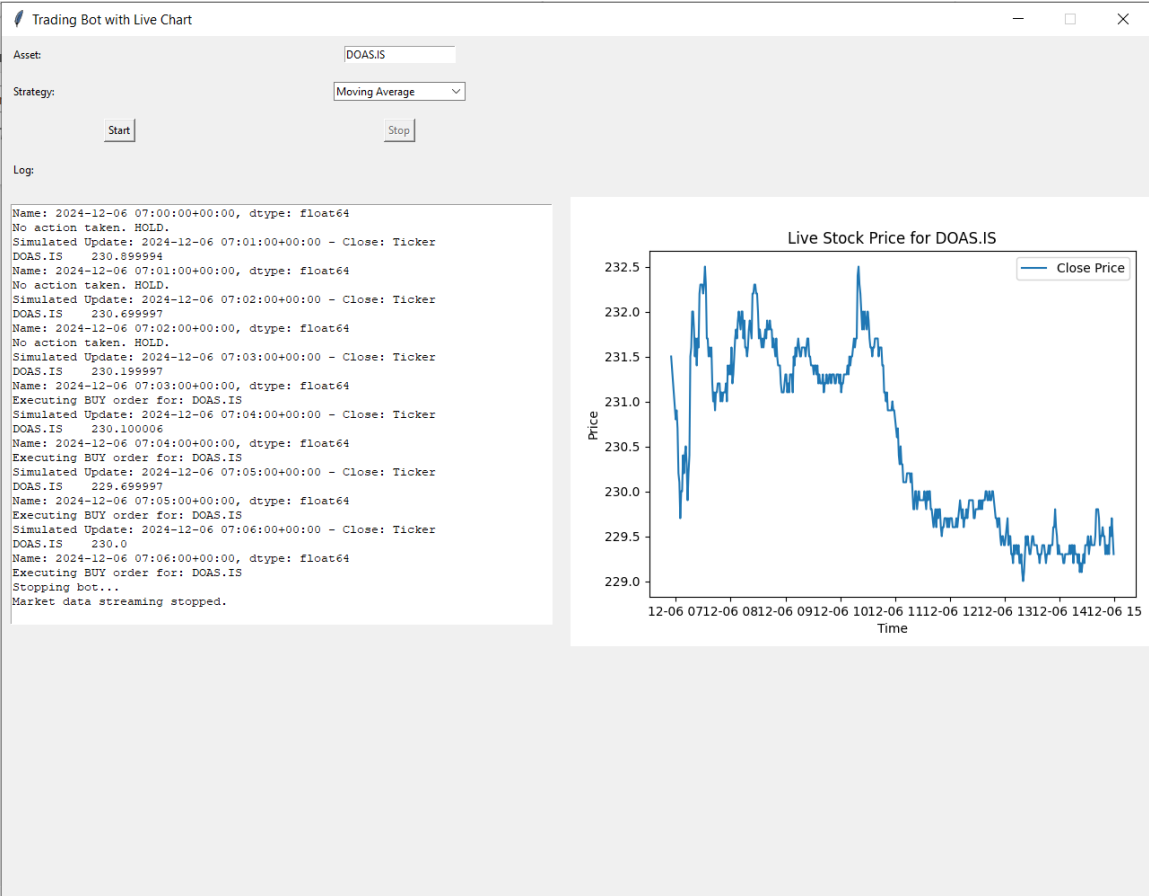
Explanation of the Code

- **MarketDataProvider:** This class fetches market data and notifies its observers whenever there is new data available.
- **MarketDataObserver:** Each observer, such as TradingBot, implements the `on_market_data_update` method to handle the updated market data when notified by the subject.

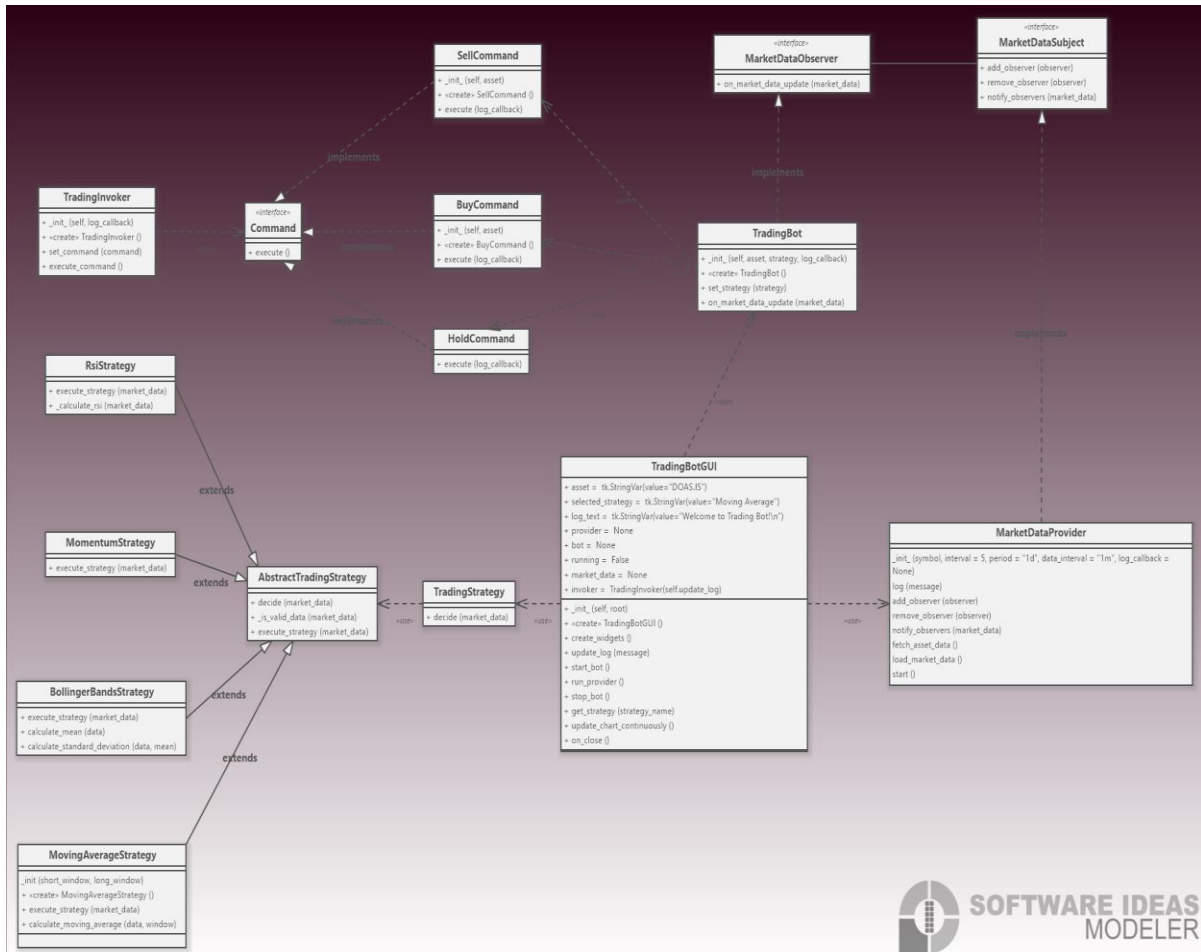
4.4 User Interface

Example for Moving Average strategy:





5. UML Diagram Explanation



Key Components:

1. MarketDataProvider (Subject)

- The MarketDataProvider class is a subject in the Observer Pattern. It is responsible for fetching the market data (using an external service like yfinance) and notifying its observers when there is an update.
- It has the methods add_observer, remove_observer, and notify_observers to manage its observers.
- It is linked to the TradingBot through the add_observer method.

2. MarketDataObserver (Observer)

- This is an abstract class that defines the on_market_data_update method, which is called whenever the market data is updated.
- TradingBot implements this interface and reacts to the market data updates by making trading decisions.

3. TradingBot (Observer)

- The TradingBot class implements the MarketDataObserver interface. This means it receives updates from the MarketDataProvider and reacts accordingly based on the selected trading strategy.
- It holds a reference to a strategy (such as MomentumStrategy, BollingerBandsStrategy, etc.) and uses this to decide whether to buy, sell, or hold assets.

4. TradingInvoker

- This class is used to invoke trading commands. It holds a reference to the Command (Buy, Sell, Hold commands) and executes them. It's essentially a helper class that decouples the TradingBot from the actual execution of the orders.

5. AbstractTradingStrategy (Abstract Class)

- The AbstractTradingStrategy class defines the base class for all trading strategies.
- It provides the decide method (a Template Method) which calls execute_strategy — a method that must be implemented by all subclasses. The decide method checks if the market data is valid before executing the specific strategy.

6. Concrete Strategy Classes:

- These are classes that implement specific trading strategies. Examples include:
 - **MomentumStrategy:** This strategy buys when the last price is higher than the second-to-last price and sells when the last price is lower.
 - **BollingerBandsStrategy:** This strategy uses Bollinger Bands to make decisions, buying when the price is below the lower band and selling when it's above the upper band.
 - **RsiStrategy:** This strategy buys when the RSI (Relative Strength Index) is below 30 and sells when it's above 70.
 - **MovingAverageStrategy:** This strategy compares short and long moving averages to make decisions.

7. Command Interface and Concrete Commands:

- The Command Pattern is used to encapsulate trading actions (Buy, Sell, Hold) into command objects.
- The BuyCommand, SellCommand, and HoldCommand classes implement the Command interface and define the execute method.
- These commands are invoked by the TradingInvoker to carry out the respective actions.

Relationships:

1. MarketDataProvider -> MarketDataObserver:

- MarketDataProvider notifies its observers (like the TradingBot) about updates in market data. The TradingBot reacts to these updates by executing the appropriate strategy.

2. TradingBot -> TradingStrategy:

- The TradingBot holds a reference to a TradingStrategy. It uses this strategy to make buy, sell, or hold decisions based on the market data received.

3. TradingBot -> TradingInvoker -> Command:

- The TradingBot uses TradingInvoker to execute commands. The Command Pattern allows the actions (buy, sell, hold) to be encapsulated in command objects, making the system more flexible and decoupled.

4. TradingInvoker -> Command (Buy, Sell, Hold):

- TradingInvoker is responsible for invoking the execute method of the command objects. These commands represent the trading actions that the bot performs.

5. Concrete Strategies -> AbstractTradingStrategy:

- The various trading strategies (like MomentumStrategy, BollingerBandsStrategy) extend AbstractTradingStrategy, implementing their specific logic in the execute_strategy method.

Diagram Summary:

- The diagram effectively demonstrates how the Observer Pattern and Command Pattern are applied to a trading bot system. The Observer Pattern allows for updates from market data to trigger changes in the trading bot's behavior, while the Command Pattern ensures that trading actions (buy, sell, hold) are decoupled from the bot's core logic and are encapsulated as command objects.
- The Template Method Pattern is represented in the AbstractTradingStrategy, where the general process (like validating data) is fixed, and only the specific strategy logic (buy/sell/hold) is flexible.
- The Strategy Pattern is used to provide interchangeable trading strategies that can be switched dynamically by the TradingBot.

6. Results and Future Work

Results

The project successfully implements a flexible trading bot that can dynamically switch between different trading strategies, execute buy, sell, and hold commands, and monitor market data changes in real time.

Future Work

Future improvements may include:

- Implementing real-time trading functionality.
- Adding more advanced trading strategies.
- Integrating AI-based analysis for better decision-making.
- Enabling multi-strategy trading simultaneously.

Zehra Selin Karabıçak 20220808621

Berat Kargin 20210808064

İsmail Özkan 20210808011