Stock Trading Application Design using Design Patterns

# Question:

Suppose you're asked to create a system for a stock trading application. Traders can subscribe to price updates of certain stocks (Observer Pattern). Depending on the type of trade (e.g., market order, limit order), different algorithms might be used to execute trades (Strategy Pattern). A trading engine logs every action taken and ensures only one instance of the logger exists (Singleton Pattern). Different types of trades might also require creating specific trade processors (Factory Pattern). How would you design this system?

# Solution Approach:

- Observer Pattern: Implement a Stock class that acts as the Subject. Traders are Observers who get notified when there is a price change.  
 - Strategy Pattern: Define a TradeStrategy interface with different trade execution strategies, such as MarketOrderStrategy and LimitOrderStrategy.  
 - Singleton Pattern: Use Singleton for the Logger class, ensuring that all trade actions are logged to a central logging instance.  
 - Factory Pattern: Implement a TradeProcessorFactory to create different types of TradeProcessor objects based on the trade type.

## 1. Adding stockSymbol for Multiple Stocks (Observer Pattern):

In the Stock class, we add a String stockSymbol field to uniquely identify each stock. Each Trader observer can then receive updates that specify which stock’s price has changed.

### Observer Pattern Code:

**2. Observer Pattern: Real-Time Notifications**

"To provide real-time price updates to traders, I implemented the Observer Pattern. In this design, each Trader acts as an observer to Stock price changes. Whenever a Stock price updates, it notifies all its registered Trader observers. This pattern allowed me to decouple the Trader class from the Stock class while ensuring that any trader interested in a stock receives immediate updates.

"I chose the Observer Pattern because it allows flexibility in managing subscribers and notifications without tightly coupling stock and trader logic. This makes it easy to add or remove traders without impacting the stock price update logic. For example, a Trader can be subscribed or unsubscribed dynamically, keeping the system highly adaptable and scalable."

// Subject Interface  
public interface Subject {  
 void attach(Observer observer);  
 void detach(Observer observer);  
 void notifyObservers();  
}  
  
// Stock Class  
public class Stock implements Subject {  
 private String stockSymbol;  
 private double price;  
 private List<Observer> observers = new ArrayList<>();  
  
 public Stock(String stockSymbol, double price) {  
 this.stockSymbol = stockSymbol;  
 this.price = price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 notifyObservers();  
 }  
  
 public String getStockSymbol() {  
 return stockSymbol;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 @Override  
 public void attach(Observer observer) {  
 observers.add(observer);  
 }  
  
 @Override  
 public void detach(Observer observer) {  
 observers.remove(observer);  
 }  
  
 @Override  
 public void notifyObservers() {  
 for (Observer observer : observers) {  
 observer.update(stockSymbol, price);  
 }  
 }  
}  
  
// Observer Interface  
public interface Observer {  
 void update(String stockSymbol, double price);  
}  
  
// Trader Class (Observer)  
public class Trader implements Observer {  
 private String name;  
  
 public Trader(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void update(String stockSymbol, double price) {  
 System.out.println("Trader " + name + " notified: " + stockSymbol + " price changed to $" + price);  
 }  
}

## 2. Adding Trade Execution Strategies (Strategy Pattern):

**Strategy Pattern: Trade Execution**

"Since the application supports multiple trade types, each with a different execution algorithm (e.g., market order, limit order), I applied the Strategy Pattern. This pattern helped encapsulate each trading algorithm as a strategy, allowing the trading system to switch between strategies at runtime based on trade type.

"For instance, if a market order is selected, the MarketOrderStrategy executes the trade immediately at the current market price, whereas a LimitOrderStrategy might wait for a specific price. The Strategy Pattern is ideal here because it encapsulates each trade logic independently, enabling easier testing and extensibility if new trade strategies are added."

The Strategy Pattern is used to encapsulate different trading strategies based on the type of trade. A TradeExecutionContext is used to set the trade strategy at runtime.

// TradeStrategy Interface  
public interface TradeStrategy {  
 void executeTrade(String stockSymbol, double price);  
}  
  
// MarketOrderStrategy Class  
public class MarketOrderStrategy implements TradeStrategy {  
 @Override  
 public void executeTrade(String stockSymbol, double price) {  
 System.out.println("Executing market order for " + stockSymbol + " at $" + price);  
 }  
}  
  
// LimitOrderStrategy Class  
public class LimitOrderStrategy implements TradeStrategy {  
 private double targetPrice;  
  
 public LimitOrderStrategy(double targetPrice) {  
 this.targetPrice = targetPrice;  
 }  
  
 @Override  
 public void executeTrade(String stockSymbol, double price) {  
 if (price <= targetPrice) {  
 System.out.println("Executing limit order for " + stockSymbol + " at $" + price);  
 } else {  
 System.out.println("Limit order for " + stockSymbol + " pending until price reaches $" + targetPrice);  
 }  
 }  
}  
  
// TradeExecutionContext Class  
public class TradeExecutionContext {  
 private TradeStrategy strategy;  
  
 public void setTradeStrategy(TradeStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public void executeTrade(String stockSymbol, double price) {  
 if (strategy != null) {  
 strategy.executeTrade(stockSymbol, price);  
 } else {  
 System.out.println("Trade strategy not set.");  
 }  
 }  
}

## 3:Logger

**Singleton Pattern: Centralized Logging**

"To maintain a centralized log of all trading actions, I used the Singleton Pattern for the logging mechanism. The Singleton ensures that only one instance of the Logger exists, so every part of the application logs actions to a single, unified source, preventing issues like multiple log entries or duplication in a distributed system.

"I chose the Singleton Pattern here because it ensures resource control and consistency—especially important for logging. By having a single Logger instance, I avoid potential synchronization problems and ensure that all logs are consistent and accessible from one place."

**public** **class** Logger {

**private** **static** Logger *insatnce*;

**private** Logger() {

// **TODO** Auto-generated constructor stub

}

**public** **static** Logger getInsatnce()

{

**if**(*insatnce* ==**null**)

{

*insatnce* = **new** Logger();

}

**return** *insatnce*;

}

**public** **void** log(String message)

{

System.***out***.println( "log : " +message);

}

}

# 4: TradeProcessor

**5. Factory Pattern: Trade Processor Creation**

"Finally, I used the Factory Pattern to handle different types of trade processors, such as MarketOrderProcessor or LimitOrderProcessor, depending on the trade type. The Factory Pattern simplifies object creation based on dynamic criteria, making it easy to extend the application by adding new processors without modifying the core creation logic.

"I chose the Factory Pattern to avoid conditional logic when creating trade processors. Instead of a complex if-else structure to determine the processor type, I used a TradeProcessorFactory, which returns the appropriate processor instance based on the input trade type. This choice improves readability, maintainability, and scalability

**public** **interface** TradeProcessor {

**void** processTrade(**double** ammount);

}

**public** **class** LimitTradeProcessor **implements** TradeProcessor {

@Override

**public** **void** processTrade(**double** ammount) {

// **TODO** Auto-generated method stub

System.***out***.println("Processing limit order for amount: " + ammount);

}

}

**public** **class** MarketTradeProcessor **implements** TradeProcessor {

@Override

**public** **void** processTrade(**double** amount) {

System.***out***.println("Processing market order for amount: " + amount);

// Logic to execute a market order

}

}

**public** **class** TradeProcessorFactory {

**public** TradeProcessor createTradeProcsser(String tradeType)

{

**if**("MArket".equalsIgnoreCase(tradeType))

{

**return** **new** MarketTradeProcessor();

}

**else** **if**("LIMIT".equalsIgnoreCase(tradeType))

{

**return** **new** LimitTradeProcessor();

}

**throw** **new** IllegalArgumentException("Unknown trade type: " + tradeType);

}

}

# 5: StockTradingApp

**public** **class** StockTradingApp {

**public** **static** **void** main(String[] args) {

Logger logger = Logger.*getInsatnce*();

Stock appleStock = **new** Stock("Apple", 150,"AAPL");

Observer trader1 = **new** Traders("tardeeer 1");

Observer trader2 = **new** Traders("tardeeer 2");

appleStock.addObserver(trader1);

appleStock.addObserver(trader2);

appleStock.setPreice(155);

TradeService tradeService = **new** TradeService();

tradeService.executeTrade("MARKET", 1000);

tradeService.executeTrade("LIMIT", 2000);

logger.log("Application FInished");

//Using Strategy Pattern for trade execution

TradeExecutionContext tradeContext = **new** TradeExecutionContext();

// Setting and executing a Market Order

tradeContext.setTradeStrategy(**new** MarketOrderStrategy());

tradeContext.excuteTreade("AAPL", 155.00);

// Setting and executing a Limit Order

tradeContext.setTradeStrategy(**new** LimitOrderStrategy(152.00));

tradeContext.excuteTreade("AAPL", 150.00);

}

}

## Interview Explanation and Considerations:

In an interview, explain how each pattern solves a specific requirement:  
- Observer Pattern allows for real-time stock price notifications to multiple traders, ensuring up-to-date information.  
- Strategy Pattern enables flexibility in trading execution by allowing the trading algorithm to change at runtime, improving testability and modularity.  
- Singleton Pattern ensures a centralized Logger instance, maintaining a consistent log of actions without creating multiple Logger instances.  
- Factory Pattern simplifies the creation of TradeProcessor instances for different trade types, improving code extensibility.  
  
Potential Interview Questions:  
1. Why did you choose the Observer Pattern for price updates?  
 - To allow real-time updates to multiple traders without tightly coupling them to the Stock class.  
2. How does the Strategy Pattern benefit the trade execution logic?  
 - By encapsulating algorithms as strategies, we can dynamically select the appropriate trading logic.  
3. What are the pros and cons of using these design patterns?  
 - Pros: Modularity, scalability, maintainability. Cons: Increased complexity, overhead of multiple classes.

**1. Understanding the Requirements**

"To design a stock trading application, the requirements included real-time price updates for subscribed traders, multiple trade execution algorithms (like market and limit orders), a unified logging mechanism for trades, and different processors to handle various types of trades. I identified several patterns that suited these requirements, each addressing specific challenges in the application."

**2. Observer Pattern: Real-Time Notifications**

"To provide real-time price updates to traders, I implemented the Observer Pattern. In this design, each Trader acts as an observer to Stock price changes. Whenever a Stock price updates, it notifies all its registered Trader observers. This pattern allowed me to decouple the Trader class from the Stock class while ensuring that any trader interested in a stock receives immediate updates.

"I chose the Observer Pattern because it allows flexibility in managing subscribers and notifications without tightly coupling stock and trader logic. This makes it easy to add or remove traders without impacting the stock price update logic. For example, a Trader can be subscribed or unsubscribed dynamically, keeping the system highly adaptable and scalable."

**3. Strategy Pattern: Trade Execution**

"Since the application supports multiple trade types, each with a different execution algorithm (e.g., market order, limit order), I applied the Strategy Pattern. This pattern helped encapsulate each trading algorithm as a strategy, allowing the trading system to switch between strategies at runtime based on trade type.

"For instance, if a market order is selected, the MarketOrderStrategy executes the trade immediately at the current market price, whereas a LimitOrderStrategy might wait for a specific price. The Strategy Pattern is ideal here because it encapsulates each trade logic independently, enabling easier testing and extensibility if new trade strategies are added."

**4. Singleton Pattern: Centralized Logging**

"To maintain a centralized log of all trading actions, I used the Singleton Pattern for the logging mechanism. The Singleton ensures that only one instance of the Logger exists, so every part of the application logs actions to a single, unified source, preventing issues like multiple log entries or duplication in a distributed system.

"I chose the Singleton Pattern here because it ensures resource control and consistency—especially important for logging. By having a single Logger instance, I avoid potential synchronization problems and ensure that all logs are consistent and accessible from one place."

**5. Factory Pattern: Trade Processor Creation**

"Finally, I used the Factory Pattern to handle different types of trade processors, such as MarketOrderProcessor or LimitOrderProcessor, depending on the trade type. The Factory Pattern simplifies object creation based on dynamic criteria, making it easy to extend the application by adding new processors without modifying the core creation logic.

"I chose the Factory Pattern to avoid conditional logic when creating trade processors. Instead of a complex if-else structure to determine the processor type, I used a TradeProcessorFactory, which returns the appropriate processor instance based on the input trade type. This choice improves readability, maintainability, and scalability."

**6. Summary of Design Choices**

"In summary, I chose these patterns because each one solves a specific problem in the application while enhancing modularity and flexibility:

* The Observer Pattern enables real-time updates in a loosely coupled way.
* The Strategy Pattern encapsulates trade execution logic, allowing it to be switched easily.
* The Singleton Pattern ensures centralized logging.
* The Factory Pattern simplifies trade processor creation and allows easy addition of new processors.

"Overall, these patterns helped build a flexible, modular application that meets the requirements and can be extended in the future."

This response addresses the requirements, design decisions, and rationale for using each pattern. Emphasizing the patterns’ specific benefits (decoupling, flexibility, scalability) and their relevance to the requirements demonstrates an understanding of design patterns and a thoughtful approach to system architecture.

+-----------------+

| Stock | <----- Subject (Observer Pattern)

+-----------------+

| - stockSymbol: String

| - price: double

| - observers: List<Observer>

+-----------------+

| + getPrice(): double

| + setPrice(price: double): void

| + attach(observer: Observer): void

| + detach(observer: Observer): void

| + notifyObservers(): void

+-----------------+

/|\ |

| |

| |

+--|--------|---+

| |

+--------------------+ +-------------------+

| Trader (Observer) | | TradeProcessor | <---- Factory Pattern

+--------------------+ +-------------------+

| - name: String | | + processTrade(): void

| + update(stockSymbol: String, price: double): void |

+--------------------+

+-----------------------+

| TradeProcessorFactory | <--- Factory Pattern

+-----------------------+

| + getTradeProcessor(type: String): TradeProcessor |

+-----------------------+

/|\ |

| |

| |

+-------|-------+

| |

+-------------------+ +-------------------+

| MarketOrderProcessor | | LimitOrderProcessor |

+-------------------+ +-------------------+

| + processTrade(): void| | + processTrade(): void|

+-------------------+ +-------------------+

+--------------------+

| Logger | <---- Singleton Pattern

+--------------------+

| - instance: Logger

+--------------------+

| + getInstance(): Logger

| + log(message: String): void

+--------------------+

+-------------------+

| TradeStrategy | <---- Strategy Pattern

+-------------------+

| + execute(): void |

+-------------------+

/|\ |

| |

| |

+-------|------+-------+

| | |

+-------------------+ +-------------------+

| MarketOrderStrategy | | LimitOrderStrategy |

+-------------------+ +-------------------+

| + execute(): void | | + execute(): void |

+-------------------+ +-------------------+