주식 거래 프로그램 설계 명세서

1. 클래스 설계

1.1 Market 클래스

역할: 주식 시장 전체를 관리하고 가격 변동을 시뮬레이션하는 핵심 환경 클래스

데이터 멤버

멤버명	타입	설명
allStocks	vector<Stock*>	상장된 모든 주식 객체 리스트
transactionHistory	vector<Transaction>	모든 거래 내역 저장

멤버 함수

함수명	반환	설명
simulateFluctuation()	void	주가 변동 시뮬레이션 (95% 정상, 5% 폭락)
getStockByName(name: string)	Stock*	이름으로 주식 검색
getMarketVolatility()	double	시장 변동성 계산 (전 종목 변동률 표준편차)
getAverageVolume()	double	전체 시장 평균 거래량 계산
addTransaction(t: Transaction)	void	거래 내역을 transactionHistory에 추가

1.2 Stock 클래스

역할: 개별 주식의 가격과 거래량 정보 관리

데이터 멤버

멤버명	타입	설명
name	string	주식 이름
currentPrice	double	현재 가격
previousPrice	double	전일 가격
volume	double	현재 거래량
averageVolume	double	평균 거래량

멤버 함수

함수명	반환	설명
updatePrice(newPrice: double)	void	가격 업데이트 및 이전 가격 기록
getFluctuationRate()	double	가격 변동률 계산 (백분율)
setVolume(newVolume: double)	void	거래량 설정 및 평균 갱신

함수명	반환	설명
getAverageVolume()	double	평균 거래량 반환
getVolumeRatio()	double	평균 대비 거래량 비율
Stock(name: string, initPrice: double, initVolume: double)	-	생성자 averageVolume은 initVolume으로 초기화

1.3 Transaction 클래스

역할: 거래 내역 기록 및 출력

데이터 멤버

멤버명	타입	설명
stockName	string	거래 주식명
type	string	매수/매도 구분 ("BUY"/"SELL")
quantity	int	거래 수량
price	double	거래 가격 (체결 단가)
totalAmount	double	총 거래 금액 (price × quantity)
isGapTrade	bool	폭락장 매수 여부 태그
timestamp	time_t	거래 시각

멤버 함수

함수명	반환	설명
printLog()	void	거래 내역 출력
getType()	string	거래 타입 반환

1.4 Position 구조체

역할: 보유 주식의 수량과 매수 정보를 관리하는 구조체

데이터 멤버

멤버명	타입	설명
stock	Stock*	보유 주식 객체 포인터
quantity	int	보유 수량
avgBuyPrice	double	평균 매수 단가
buyTimestamp	time_t	최초 매수 시각

1.5 Trader 클래스

역할: 투자자의 포트폴리오 관리 및 매매 실행

데이터 멤버

멤버명	타입	설명
username	string	사용자명
cash	double	현금 잔고
portfolio	map<string, Position>	일반 보유 주식 (종목명 → Position)
gapPositions	map<string, Position>	폭락장 매수 주식 (종목명 → Position)
detector	unique_ptr<DisruptionDetector>	시장 분석기 (스마트 포인터)

멤버 함수

함수명	반환	설명
buy(stock: string, qty: int, m: Market&)	bool	일반 매수 (성공/실패 반환)
sell(stock: string, qty: int, m: Market&)	bool	일반 매도 (성공/실패 반환)
autoTrade(m: Market&)	void	자동 거래 실행
buyGapPosition(stock: string, qty: int, m: Market&)	bool	폭락장 매수
closeAllGapPositions(m: Market&)	void	폭락장 포지션 전량 청산
closeGapPosition(stock: string, m: Market&)	bool	특정 갭 포지션 청산
hasOpenGapPositions()	bool	폭락장 포지션 보유 여부
getPositionQuantity(stock: string)	int	특정 종목 보유 수량 조회
getTotalAssetValue(m: Market&)	double	총 자산 가치 계산
getGapPositionProfit(stock: string)	double	갭 포지션 수익률 계산

1.6 DisruptionDetector 클래스

역할: 시장 위험도 분석 및 투자 전략 결정

데이터 멤버

멤버명	타입	설명
currentState	unique_ptr<MarketState>	현재 시장 상태 (스마트 포인터)
disruptionScore	double	시장 위험도 점수
THRESHOLD	const double = 70.0	폭락 판단 임계값
RECOVERY_THRESHOLD	const double = 50.0	회복 판단 임계값 (신규)
previousScore	double	이전 위험도 점수

멤버 함수

함수명	반환	설명
analyzeMarket(m: Market&)	void	시장 분석 및 자동 상태 전환
changeState(newState: MarketState*)	void	상태 전환 (메모리 자동 관리)
executeStrategy(t: Trader&, m: Market&)	void	현재 상태에 따른 전략 실행
calculateScore(m: Market&)	double	위험 점수 계산
checkRecovery()	bool	시장 회복 여부 검사
getDisruptionScore()	double	위험도 점수 반환
isDisrupted()	bool	폭락 상태 여부

1.7 MarketState 클래스 (추상)

역할: 시장 상태의 기본 인터페이스 (State Pattern)

멤버 함수

함수명	반환	설명
handle(t: Trader&, m: Market&)	void	순수 가상 함수 - 전략 실행
getStateName()	string	순수 가상 함수 - 상태명 반환
~MarketState()	virtual	가상 소멸자

1.8 NormalState 클래스

역할: 정상 시장 상태 전략

데이터 멤버

멤버명	타입	설명
TARGET_PROFIT_RATE	const double = 0.10	갭 포지션 목표 수익률 (10%)

멤버 함수

함수명	반환	설명
handle(t: Trader&, m: Market&)	void	보수적 리밸런싱, 갭 포지션 청산 검사
getStateName()	string	"Normal" 반환
checkGapPositionRecovery(t: Trader&, m: Market&)	void	목표 수익률 도달 시 청산

1.9 DisruptedState 클래스

역할: 폭락 시장 상태 전략

데이터 멤버

멤버명	타입	설명
INVESTMENT_RATIO	const double = 0.50	투자 비율 (현금의 50%)
MAX_POSITIONS	const int = 3	최대 동시 갭 포지션 수

멤버 함수

함수명	반환	설명
handle(t: Trader&, m: Market&)	void	특수 전략 실행
getStateName()	string	"Disrupted" 반환
executeGapTrading(t: Trader&, m: Market&)	void	급락 주식 매수
executeContrarianBuy(t: Trader&, m: Market&)	void	역발상 매수
selectBestTarget(m: Market&)	Stock*	최적 매수 대상 선정
calculateInvestAmount(t: Trader&, targets: int)	double	종목당 투자 금액 계산

2. 클래스 간 관계

관계 유형	표기	클래스	설명
Composition	◆—	Market → Stock	시장이 주식을 소유 (강한 생명주기)
Aggregation	◇—	Market → Transaction	시장이 거래 내역 기록 (약한 관계)
Aggregation	◇—	Trader → Position (Portfolio)	투자자가 포지션 보유
Aggregation	◇—	Trader → Position (Gap)	폭락장 매수 포지션 보유
Composition	◆—	Trader → DisruptionDetector	투자자가 분석기 소유 (unique_ptr)
Composition	◆—	DisruptionDetector → MarketState	분석기가 상태 소유 (unique_ptr)
Dependency	..>	Trader → Market	거래 시 시장 정보 사용
Dependency	..>	Trader → Transaction	거래 발생 시 생성 후 Market에 등록
Dependency	..>	DisruptionDetector → Market	분석 시 시장 데이터 읽기

관계 유형	표기	클래스	설명
Inheritance	——▷	NormalState → MarketState	상속 (가상 소멸자 포함)
Inheritance	——▷	DisruptedState → MarketState	상속 (가상 소멸자 포함)

3. 핵심 알고리즘

3.1 시장 변동 시뮬레이션

확률	변동 범위
95%	-2% ~ +2% (정상 등락)
5%	-10% ~ -30% (블랙 스완 이벤트)

3.2 평균 거래량 계산

계산 방식 : 지수이동평균 (EMA)

```
averageVolume = averageVolume × 0.95 + newVolume × 0.05
```

갱신 시점:

Stock::setVolume() 호출 시 자동 갱신. 별도의 이력 저장없이 누적 평균으로 관리하여 메모리 효율적.

3.3 시장 변동성 계산

정의:

시장 변동성은 전체 종목 변동률의 표준편차로 정의

```
volatility = √(Σ(각 종목 변동률 - 평균 변동률)² / N)
```

반환값 범위:

0.0 ~ 1.0 사이의 실수값 (예: 0.05 = 5% 변동성)

3.4 위험도 계산 공식

```
DisruptionScore = (현재거래량 / 평균거래량) × 변동성 × 1000
```

계산 예시:

거래량 비율 = 2.5 (평균의 250%), 변동성 = 0.08 (8%) → Score = 2.5 × 0.08 × 1000 = 200 → Disrupted

거래량 비율 = 1.2, 변동성 = 0.03 → Score = 1.2 × 0.03 × 1000 = 36 → Normal

상태 판단 기준:

조건	결과
Score > 70.0 (THRESHOLD)	DisruptedState 전환
Score < 50.0 (RECOVERY_THRESHOLD)	NormalState 복귀

조건	결과
50.0 ≤ Score ≤ 70.0	현재 상태 유지

3.5 스마트 역발상 매수 전략

발동 조건:

조건	값
가격 하락률	< -15%
거래량	> 평균 거래량 × 200%
현재 갭 포지션 수	< MAX_POSITIONS (3개)

매수 대상 선정 기준 (selectBestTarget):

우선순위	기준	설명
1	하락률 최대	가장 많이 하락한 종목 우선
2	거래량 비율 최대	동일 하락률 시 거래량 폭증 종목
3	기존 미보유	이미 갭 포지션 보유 종목 제외

투자 금액 계산 (calculateInvestAmount):

```
종목당 투자금액 = (현금 × INVESTMENT_RATIO) / 조건충족_종목수
```

예시:

현금 1,000만원, 조건 충족 종목 2개 → 종목당 250만원 투자 (1000 × 0.5 / 2)

3.6 갭 포지션 청산 조건

자동 청산 조건:

조건 유형	세부 내용
목표 수익 달성	수익률 ≥ TARGET_PROFIT_RATE (10%) 시 해당 포지션 청산
시장 정상화	Score < RECOVERY_THRESHOLD (50.0) 상태에서 수익 포지션 순차 청산
손절 (선택)	수익률 < -20% 시 손절 (구현 선택사항)

수익률 계산:

```
profitRate = (현재가 - 평균매수가) / 평균매수가
```

4. 거래 처리 흐름

4.1 매수 처리 흐름

1. Trader::buy() 호출 → 2. 잔고 확인 → 3. Stock 가격 조회 → 4. Transaction 생성 → 5. Market::addTransaction() 호출 → 6. Portfolio 업데이트 → 7. 현금 차감

4.2 매도 처리 흐름

1. Trader::sell() 호출 → 2. 보유 수량 확인 → 3. Stock 가격 조회 → 4. Transaction 생성 → 5. Market::addTransaction() 호출 → 6. Portfolio 업데이트 → 7. 현금 증가

4.3 자동 거래 흐름

1. Trader::autoTrade() 호출 → 2. DisruptionDetector::analyzeMarket() → 3. 상태 자동 전환 판단 → 4. DisruptionDetector::executeStrategy() → 5. 현재 State의 handle() 실행 → 6. 필요시 매수/매도 수행

5. 메모리 관리 지침

클래스	포인터 타입	관리 방식
Market → Stock	vector<Stock*>	Market 소멸자에서 delete
Trader → DisruptionDetector	unique_ptr	자동 해제
DisruptionDetector → MarketState	unique_ptr	changeState() 시 자동 교체
Position → Stock	Stock* (약한 참조)	해제 책임 없음 (Market 소유)

주의사항: Position 내의 Stock 포인터는 Market이 소유한 객체를 참조만 하므로, Position에서 delete하면 안된다.