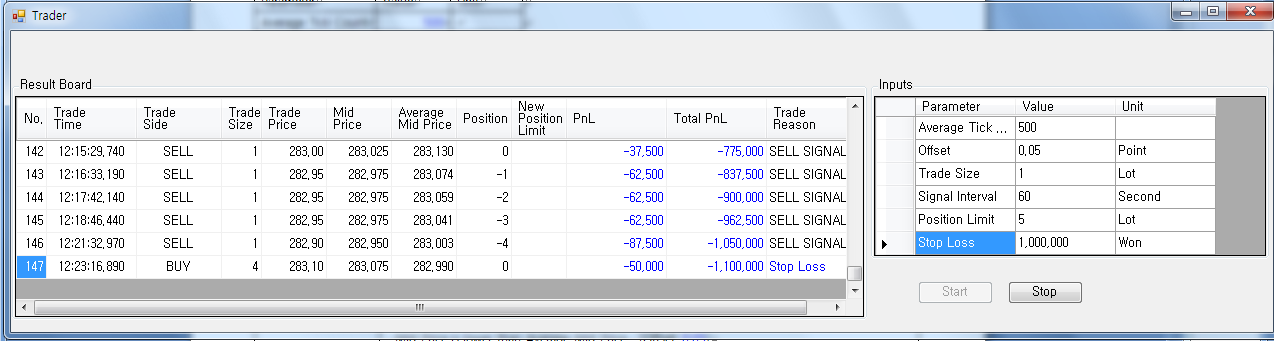
Future Trader

# Output Screenshots



# 사용법

Start 버튼을 누르면 동작을 시작한다.

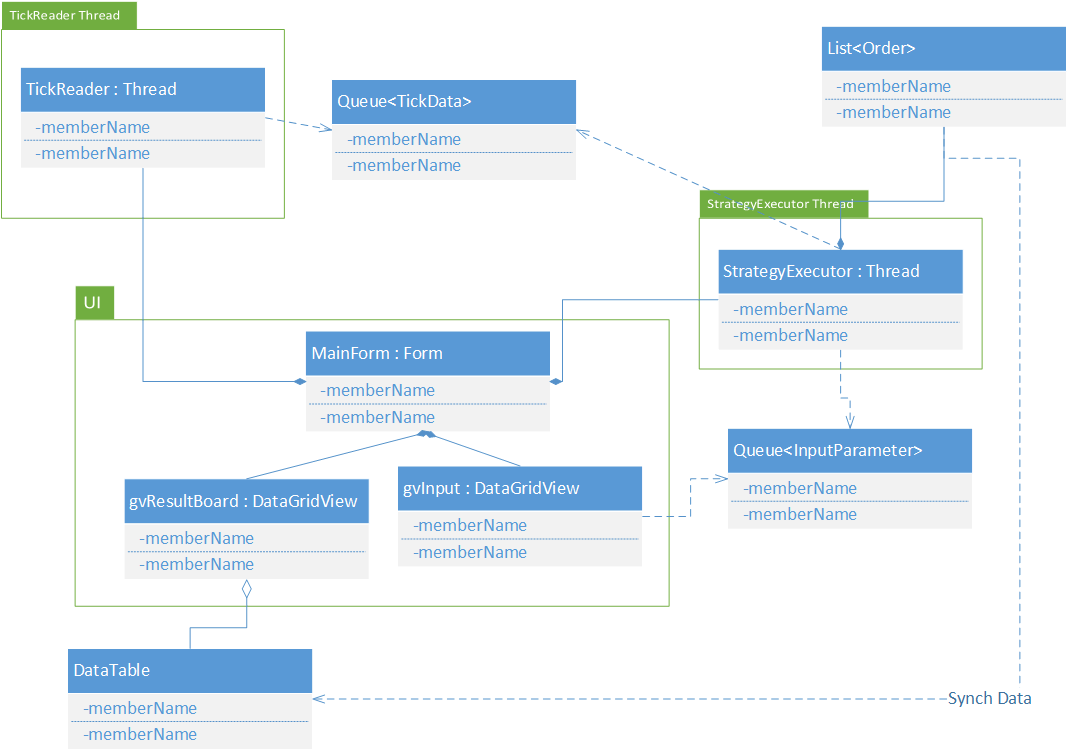
Stop 버튼을 누르면 TickReader 와 StrategyExecutor 2개의 Thread 를 pause 하게 된다. 2개의 Thread 모두 누른 시점의 loop 을 마치고 Tick 까지 계산을 마치고 Thread 를 pause 하게 된다.

다시 start 버튼을 누르면 pause 시점부터 다시 동작을 시작한다.

처음부터 시작하려면, 프로그램을 재 실행 해야 한다.

Input 을 더블클릭 하면 변경할 수 있다. Input 의 변경은 아무때나 가능하다. 변경되는 대로 전달이 되긴 하지만 정확히 어떤 tick 에 적용이 될 지는 예측 할 수 없다. 하지만, stop button 을 누른 후 변경하고 start 를 했다면, 그 start 되는 시점의 tick 에 적용될 것이다.

# Diagram



# Souce code 설명

## MainThread

UI Thread 이다. 2개의 Thread 를 실행시키며, “거래 내역에 대한 DataGridView (gvResultBoard)” 와 “input 값을 가지고 있는 DataGridView(gvInput)” 을 생성한다.

그리고, Thread 사이의 통신을 위한 Queue 를 2개 가지고 있다. mQueue 는 TickReader 에서 읽어드린 data 를 StrategyExecutor 에 전달하기 위한 Queue 이며, mInputQueue 는 변경된 Input 값을 StrategyExecutor에 전달하기 위한 Queue 이다.

## 2개의 Thread

1. TickReader : .csv 로 되어 있는 Tick data 를 읽어오고, Queue를 통해 StrategyExecutor 에게 보낸다.(publish)
2. StrategyExecutor : TickReader 에서 받은 data 를 가지고 signal 을 분석해서 거래를 실행한다. 수행된 주문(Order) 를 orderList 에 보관하며, 이 값을 UI Thread 의 DataTable 과 동기화한다.

## Order

Trade 가 발생했을 때(Signal 이 발생하고, Signal 이 조건에 맞을 때)를 Order 라고 지칭한다.

## StrategyExecutor

변경된 input이 있는지 여부를 확인하고, 변경된 사항이 있으면 반영한다.

TickData 를 읽어온다. Tick 은 한개씩 순차적으로 처리한다. 한 번에 하나의 Tick 만을 가져오고, 하나의 Tick 에 대한 처리가 끝난 후 다른 하나를 다시 가져와서 처리한다.

매 Tick 마다 현재 Tick 의 Mid 값으로 Total PnL 을 계산해서 UI 에 update 를 시켜준다.

TickReader 와 AutoResetEvent() 를 공유하고 있으며, 이 Event 로 TickReader 가 실행된 이후에 실행하는 것을 보장 받게 된다.

그리고 pause 를 위한 ManualResetEvent() 이 있다. TickReader 에서 전달 된 값을 Dequeue 하는 loop 앞에 존재하며 Stop 버튼이 눌렸을 때 다음 tick 에서 StrategyExecutor 의 동작을 멈추게 한다.

### 종료 조건, Stop Loss

Stop Loss 의 값에 도달하면(손해가 Stop loss 와 같거나 초과하는 경우), 현재의 market price 로 남아있는 포지션을 처리(SELL 또는 BUY)해서 Position 을 ‘0’ 으로 만들고, 마지막 row 에 “Stop loss” 로 발생한 Trade 를 추가하고, loop 을 중단한다.

## Signal 분석

### Average Mid Price

현재 tick data 를 “포함”해서 이전 500개(Average Tick Count) data 에 대한 평균값을 구한다.

구하는 공식은 각 tick 의 Mid Price 를 모두 더해서 Average Tick Count 로 나누어 준다.

|  |
| --- |
| Average Mid Price = Average Tick Count 동안의 Mid Price 의 합 / Average Tick Count |

화면에 표시되는 값은 소수 3번째자리 까지만 보여준다. 소수 4번째 자리에서 반올림 해서 보여준다.

Signal 이 발생한 후, Signal 이 Accept 하는 조건하에 있다면, orderList 에 Order 를 추가한다. 그리고 나서 UI Thread 에 있는 DataTable 에 추가해 준다.

StrategyExecutor thread 에 있는 orderList와 UI Thread 에 있는 orderDataTable 은 언제나 같은 값을 유지 하도록 한다.

Total 은 계산의 정확도를 위해 orderList 에서는 decimal type 으로 유지하며, UI 에 보여주는 결과 값인 DataTable 에 추가할 때만, “50만”(Order.PRICEPERPOINT) 을 곱해서 보여준다.

### Signal 발생

Average Tick Count 의 수 만큼의 data 가 있어야 Average Mid Price 를 구할 수 있기 때문에, Average Tick Count 번째 Tick 부터 Signal 이 발생된다. 예를들어 Average Tick Count 가 500인 경우에는 Average Mid Price 를 구할 수 있는 500 번째 Tick 부터 signal 을 발생시키게 된다.

### Signal 의 Acceptance 조건(isOrderSignal() 참고)

1. Position limit 을 초과하지 않아야 한다.(아래 Position limit 참고)
2. 이전 signal 이 없는 경우(첫 signal)
3. 이전에 Accept 한 signal 과 현재 발생한 signal 이 다른 경우
4. 이전에 Accept 한 signal 과 현재 발생한 signal 이 같은 경우, 이전에 Accept 한 signal이 발생한 시간과 현재 signal 이 발생한 시간과의 차이가 Signal Interval 를 초과해야 한다.

### Position Limit

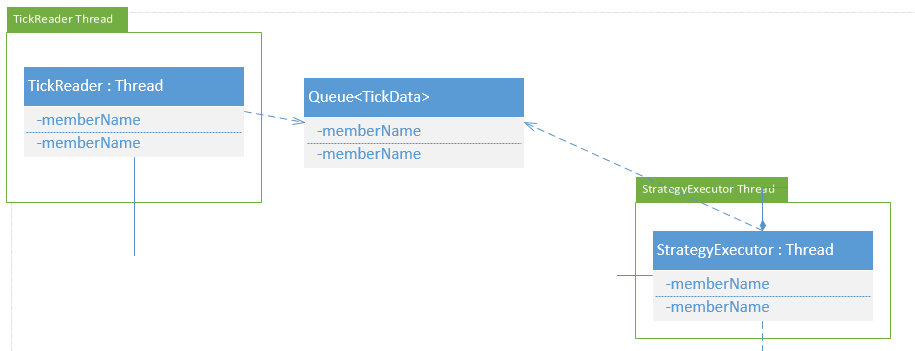
Sell signal 의 position 은 음수로 표현되고, Buy Signal 은 양수로 표현된다. Signal 에 의해 Buy 또는 Sell 의 Position 을 취했을 때 현재 취한 position 의 개수(절대값)이 Position Limit 을 넘지 않도록 했다.

| position | <= position limit

## Thread 간 통신

### TickReader <-------> StrategyExecuter

TickReader 에서 읽어드린 TickData 를 StrategyExecutor 에 전달하기 위해 Queue 를 사용한다. 그리고 Queue 가 비어 있거나, data 를 읽는 과정이 너무 오래 걸리는 경우에 StrategyExecuter thread 가 loop 을 계속해서 도는 것을 방지하기 위해 AutoResetEvent() 를 이용하고 있다.



### StrategyExecutor <-------> DataGridView

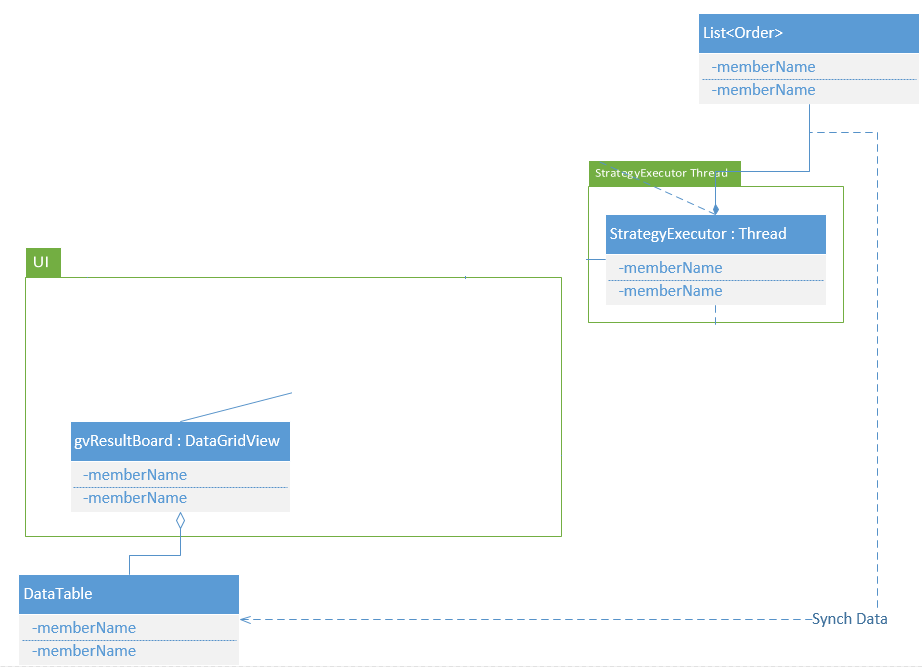
StrategyExecutor 가 가지고 있는 orderList 와 UI DataGridView 의 Binding Data 인 orderDataTable 의 값을 항상 같게 유지한다.

2개의 Data 의 동기화는

1. Order 가 추가되는 작업
2. Total PnL 의 계산결과

에 대해서 이루어 진다.

동기화의 방법은 단순하다. orderList 의 값이 update 될 때 orderDataTable 도 update 를 같이 해준다.



### Input DataGridView <-------> StrategyExecutor

Queue 를 사용해서 서로간에 값을 전달한다.

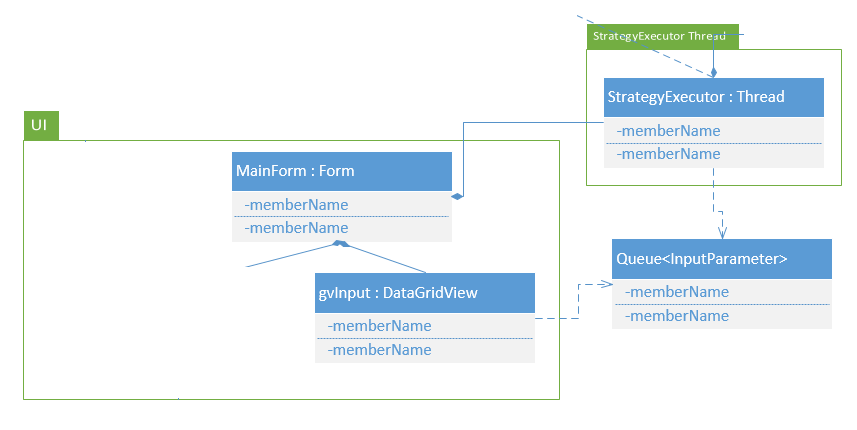
Input 이 변경된 후에 CellValidating event 가 발생하는 데 이 Event Handler 에서 변경된 input 값을 Queue 에 넣어준다.

Strategy Executor 에서는 매 Tick 마다 변경된 input 이 있는지를 확인하고 처리한다.

#### InputParameter

둘 사이의 전달하는 message 의 format 을 InputParameter 로 정의 했다.

|  |
| --- |
| Member variables |
| 1. which :어떤 input 이 변경되었는지(which) 2. Value : 변경된 값 |



## 버튼 추가

Loop 안에서 Event.WaitOne() 을 추가했다. 그래서 Stop 버튼에 의해 Loop 을 멈출 수 있도록 하였다. Stop button을 누르게 되면, TickReader 와 StrategyExecutor 2개의 Thread 의 pause 를 호출하게 된다. 이 때 Thread 내부의 event 를 Reset() 하게 되고, Event.WaitOne() 에서 blocking 되게 된다.

### StrategyExecutor 의 WaitHandle

TickReader 가 pause 된 후 queue 에 쌓여있는 data 를 처리하려고, 오랫동안 loop 을 돌게 된다.그래서 StrategyExecutor 에도 WaitHandle 를 추가해, stop button 이 누르면 바로 loop 을 멈출 수 있게 하였다.

## Result Board 의 column 추가

### New Position Limit

이 값은 StrategyExecutor 가 가지고 있는 orderList 에는 저장되지 않는다. 그저 바뀐 시점만 보여주는 역할을 한다.

### PnL

현재 row 의 현재 가격에 대한 PnL(Profit and Loss) 이다.

## Decimal 의 사용

계산의 정확도를 위해 decimal 을 사용하였다. Data 에 있는 Price 와 관계된 모든 값의 계산은 decimal 로 진행하였다. Stop loss 값의 경우는 1 point 가격인 “50만” 으로 나눈 후 사용하였다.

UI 에서 Mid Price 와 Average Mid Price 처럼 소수점 아래 일정부분까지만 보여줄때는 반올림해서 보여준다.