EPQ Artefact – MetaTrader 4 Program Code

```
input int MagicNumber = 19259;
input double PipPointOverride = 0;
double PipPoint = 0;
input bool OverrideIndicatorInputValues = 0;
string stopLossComment;
string takeProfitComment;
bool StopEA = false;
int UnitsOneLot = 100000;

enum ORDER_GROUP_TYPE { Single=1, SymbolOrderType=2, Basket=3, SymbolCode=4 };
enum ORDER_PROFIT_CALCULATION_TYPE { Pips=1, Money=2, EquityPercentage=3 };

void SetPipPoint()
{
  if (PipPointOverride != 0)
  {
    PipPoint = PipPointOverride;
  } else
  {
    PipPoint = GetRealPipPoint(Symbol());
  }

  Print("Point: " + DoubleToStr(PipPoint, 5));
}

// Pip Point Function
double GetRealPipPoint(string Currency)
{
        double calcPoint = 0;
        double calcDigits = MarketInfo(Currency, MODE_DIGITS);
        Print("Digits: " + DoubleToString(calcDigits));
```

```
        if(calcDigits == 2)
        {
          calcPoint = 0.1;
        }
        else if (calcDigits == 3)
        {
                calcPoint = 0.01;
   }
        else if(calcDigits == 4 || calcDigits == 5)
                calcPoint = 0.0001;

        return(calcPoint);
}


class Order
{
        public:
                int Ticket;
                int Type;
                int MagicNumber;
                double Lots;
                datetime OpenTime;
                double OpenPrice;
                datetime CloseTime;
                double ClosePrice;
                double StopLoss;
                double TakeProfit;
                datetime Expiration;
                double CurrentProfitPips;
                double HighestProfitPips;
                double LowestProfitPips;
                string Comment;
                double Commission;
```

```
string SymbolCode;


void Order()
{
}


void Order(Order* order)
{
		Ticket = order.Ticket;
		Type = order.Type;
		MagicNumber = order.MagicNumber;
		Lots = order.Lots;
		OpenTime = order.OpenTime;
		OpenPrice = order.OpenPrice;
		CloseTime = order.CloseTime;
		ClosePrice = order.ClosePrice;
		StopLoss = order.StopLoss;
		TakeProfit = order.TakeProfit;
		Expiration = order.Expiration;
		CurrentProfitPips = order.CurrentProfitPips;
		HighestProfitPips = order.HighestProfitPips;
		LowestProfitPips = order.LowestProfitPips;
		Comment = order.Comment;
		Commission = order.Commission;
		SymbolCode = order.SymbolCode;
}


double CalculateProfitPipettes()
{
		switch (Type)
		{
	case OP_BUY:
	   return (Bid - OpenPrice);
	   break;
```

```
      case OP_SELL:
         return (OpenPrice - Ask);
         break;
            }

   return 0;
}

double CalculateProfitPips()
{
   double pipettes = CalculateProfitPipettes();
   double pips = pipettes / PipPoint;
   return pips;
}

double CalculateProfitCurrency()
{
   switch (Type)
            {
      case OP_BUY:
         return (Bid - OpenPrice) * (UnitsOneLot * Lots) - Commission;
         break;
      case OP_SELL:
         return (OpenPrice - Ask) * (UnitsOneLot * Lots) - Commission;
         break;
            }
   return 0;
}

double CalculateProfitEquityPercentage()
{
   switch (Type)
            {
      case OP_BUY:
         return 100 * ((Bid - OpenPrice) * (UnitsOneLot * Lots) - Commission) / AccountBalance();
```

```
            break;
        case OP_SELL:
            return 100 * ((OpenPrice - Ask) * (UnitsOneLot * Lots) - Commission) / AccountBalance();
            break;
                }
    return 0;
}

double CalculateValueDifferencePips(double value)
{
    double divOpenPrice = 0.0;
    switch (Type)
    {
        case OP_BUY:
            divOpenPrice = (value - OpenPrice);
            break;

        case OP_SELL:
            divOpenPrice = (OpenPrice - value);
            break;
    }

    double pipsDivOpenPrice = divOpenPrice / PipPoint;
    return pipsDivOpenPrice;
}

void ~Order()
{
}
};

//
//  OrderCollection
//
class OrderCollection
```

```
{
    private:
        Order* _orders[];
        int _pointer;
        int _size;

    public:
        void OrderCollection()
        {
            _pointer = -1;
            _size = 0;
        }

        void ~OrderCollection()
        {
            for (int i = 0; i < ArraySize(_orders); i++)
            {
                delete(_orders[i]);
            }
        }

        void Add(Order* item)
        {
            _size = _size + 1;
            ArrayResize(_orders, _size, 8);

            _orders[(_size - 1)] = item;
        }

        Order* Remove(int index)
        {
            Order* removed = NULL;

            if (index >= 0 && index < _size)
            {
```

```
      removed = _orders[index];

      for (int i = index; i < (_size - 1); i++)
      {
         _orders[i] = _orders[i + 1];
      }

      ArrayResize(_orders, ArraySize(_orders) - 1, 8);
      _size = _size - 1;
   }

   return removed;
}

Order* Get(int index)
{
   if (index >= 0 && index < _size)
   {
      return _orders[index];
   }

   return NULL;
}

int Count()
{
   return _size;
}

void Rewind()
{
   _pointer = -1;
}

Order* Next()
```

```
{
    _pointer++;
    if (_pointer == _size)
    {
        Rewind();
        return NULL;
    }

    return Current();
}

Order* Prev()
{
    _pointer--;
    if (_pointer == -1)
    {
        return NULL;
    }

    return Current();
}

bool HasNext()
{
    return (_pointer < (_size - 1));
}

Order* Current()
{
    return _orders[_pointer];
}

int Key()
{
    return _pointer;
```

```
      }

                        int GetKeyByTicket(int ticket)
                        {
                           int keyFound = -1;
                           for (int i = 0; i < ArraySize(_orders); i++)
                           {
                              if (_orders[i].Ticket == ticket)
                              {
                                 keyFound = i;
                              }
                           }
                           return keyFound;
                        }

                        int CountByOrderType(int orderType)
                        {
                           int counter = 0;
                           for (int i = 0; i < ArraySize(_orders); i++)
                           {
                              if (_orders[i].Type == orderType)
                              {
                                 counter++;
                              }
                           }
                           return counter;
                        }

};

//
//  OrderRepository
//
class OrderRepository
{
```

```
private:
  static Order* getByTicket(int ticket)
  {
    bool orderSelected = OrderSelect(ticket, SELECT_BY_TICKET);
    if (orderSelected)
    {
      Order* order = new Order();
      OrderRepository::fetchSelected(order);
      return order;
    }
    else
    {
      return NULL;
    }
  }

  static void fetchSelected(Order& order)
  {
    order.Ticket = OrderTicket();
    order.Type = OrderType();
    order.MagicNumber = OrderMagicNumber();
    order.Lots = OrderLots();
    order.OpenPrice = OrderOpenPrice();
    order.StopLoss = OrderStopLoss();
    order.TakeProfit = OrderTakeProfit();
    order.Expiration = OrderExpiration();
    order.Comment = OrderComment();
    order.OpenTime = OrderOpenTime();
    order.CloseTime = OrderCloseTime();
  }

  static bool modify(int ticket, double stopLoss = NULL, double takeProfit = NULL)
  {
    Order* order = OrderRepository::getByTicket(ticket);
```

```
    double price = order.OpenPrice;
    stopLoss = (stopLoss == NULL)? order.StopLoss: stopLoss;
    takeProfit = (takeProfit == NULL)? order.TakeProfit: takeProfit;
    datetime expiration = order.Expiration;
    bool result = OrderModify(ticket, price, stopLoss, takeProfit, expiration);

    delete(order);
    return result;
  }

public:
  static OrderCollection* GetOpenOrders(int magic = NULL, int type = NULL)
  {
    OrderCollection* orders = new OrderCollection();

    for (int i = 0; i < OrdersTotal(); i++)
    {
      bool orderSelected = OrderSelect(i, SELECT_BY_POS);

      if (orderSelected)
      {
        Order* order = new Order();
        OrderRepository::fetchSelected(order);

        if ((magic == NULL || magic == order.MagicNumber)
          && (type == NULL || type == order.Type))
        {
          orders.Add(order);
        }
        else
        {
          delete(order);
        }
      }
    }
```

```
  return orders;
}

static int ExecuteOpenBuy(Order* order)
{
  int ticket;

  string symbol = Symbol();
  int cmd = OP_BUY;
  double price = Ask;
  int slippage = 5;
  double stoploss = order.StopLoss;
  double takeprofit = order.TakeProfit;
  string comment = order.Comment;
  ticket = OrderSend(symbol, cmd, order.Lots, price, slippage, stoploss, takeprofit, comment, MagicNumber);
  if (ticket != -1)
  {
    if(OrderSelect(ticket, SELECT_BY_TICKET) == true)
    {
      order.OpenPrice = OrderOpenPrice();
      order.OpenTime = OrderOpenTime();
    }
  }
  else
  {
    int code = GetLastError();
    if (code == 134)
    {
      Print("Not enough money detected (code 134)");
    }
  }

  return ticket;
}
```

```
static int ExecuteOpenSell(Order* order)
{
  int ticket;

  string symbol = Symbol();
  int cmd = OP_SELL;
  double price = Bid;
  int slippage = 5;
  double stoploss = order.StopLoss;
  double takeprofit = order.TakeProfit;
  string comment = order.Comment;
  ticket = OrderSend(symbol, cmd, order.Lots, price, slippage, stoploss, takeprofit, comment, MagicNumber);
  if (ticket != -1)
  {
    if(OrderSelect(ticket, SELECT_BY_TICKET) == true)
    {
      order.OpenPrice = OrderOpenPrice();
      order.OpenTime = OrderOpenTime();
    }
  }
  else
  {
    int code = GetLastError();
    if (code == 134)
    {
      Print("Not enough money detected (code 134)");
    }
  }

  return ticket;
}

static bool CloseOrder(Order* order)
{
  double price = NULL;
```

```
  switch (order.Type)
  {
    case OP_BUY:
      price = Bid;
      break;
    case OP_SELL:
      price = Ask;
      break;
  }

  if (price != NULL)
  {
    if (OrderClose(order.Ticket, order.Lots, price, 10))
    {
      bool orderSelected = OrderSelect(order.Ticket, SELECT_BY_TICKET, MODE_HISTORY);
      if (orderSelected)
      {
        order.CloseTime = OrderCloseTime();
        order.ClosePrice = OrderClosePrice();
      }

      return true;
    }
  }
  return false;
}

static Order* GetLastClosedOrder()
{
  Order* order = NULL;
  for(int i=OrdersHistoryTotal()-1;i>=0;i--)
  {
    if (OrderSelect(i, SELECT_BY_POS,MODE_HISTORY))
    {
      if(OrderSymbol()==Symbol() && OrderMagicNumber()==MagicNumber)
```

```
      {
        order = new Order();
        order.OpenTime = OrderOpenTime();
        order.CloseTime = OrderCloseTime();
        order.Type = OrderType();
        break;
      }
    }
  }
  return order;
}

static bool OpenOrder(Order* order)
{
  double price = NULL;
  int ticketId = -1;
  switch (order.Type)
  {
    case OP_BUY:
      ticketId = ExecuteOpenBuy(order);
      if (ticketId != -1)
      {
        order.Ticket = ticketId;
      }
      break;
    case OP_SELL:
      ticketId = ExecuteOpenSell(order);
      if (ticketId != -1)
      {
        order.Ticket = ticketId;
      }
      break;
  }

  return ticketId != -1;
```

```
    }
};

//
//  OrderGroupData
//
class OrderGroupData
{
        public:
                OrderCollection* OrderGroup;

                void OrderGroupData()
                {
                        OrderGroup = new OrderCollection();
                }

                void OrderGroupData(OrderGroupData* ordergroupdata)
                {
                        OrderGroup = ordergroupdata.OrderGroup;
                }

                void ~OrderGroupData()
                {
                        if (OrderGroup != NULL && CheckPointer(OrderGroup) == POINTER_DYNAMIC)
                                delete(OrderGroup);
                }
};

//
//  OrderGroupHashMap
//
class OrderGroupHashEntry
{
   public:
      string _key;
```

```
    OrderGroupData* _val;
    OrderGroupHashEntry *_next;

    OrderGroupHashEntry()
    {
      _key=NULL;
      _val=NULL;
      _next=NULL;
    }

    OrderGroupHashEntry(string key, OrderGroupData *val)
    {
      _key=key;
      _val=val;
      _next=NULL;
    }

    ~OrderGroupHashEntry()
    {
    }
};

class OrderGroupHashMap
{
  private:
    uint _hashSlots;
    int _resizeThreshold;
    int _hashEntryCount;
    OrderGroupHashEntry* _buckets[];

    bool _adoptValues;

    void init(uint size, bool adoptValues)
    {
      _hashSlots = 0;
```

```
   _hashEntryCount = 0;
   _adoptValues = adoptValues;

   rehash(size);
}

uint hash(string s)
{
   uchar c[];
   uint h = 0;

   if (s != NULL)
   {
      h = 5381;
      int n = StringToCharArray(s,c);
      for(int i = 0 ; i < n ; i++)
      {
         h = ((h << 5 ) + h ) + c[i];
      }
   }

   return h % _hashSlots;
}

uint _foundIndex;
OrderGroupHashEntry* _foundEntry;
OrderGroupHashEntry* _foundPrev;

bool find(string keyName)
{
   bool found = false;

   _foundPrev = NULL;
   _foundIndex = hash(keyName);
```

```
    if (_foundIndex <= _hashSlots)
    {
        for (OrderGroupHashEntry *e = _buckets[_foundIndex]; e != NULL ; e = e._next)
        {
            if (e._key == keyName)
            {
                _foundEntry = e;
                found=true;
                break;
            }

            _foundPrev = e;
        }
    }

    return found;
}

uint getSlots()
{
    return _hashSlots;
}

bool rehash(uint newSize)
{
    bool ret = false;
    OrderGroupHashEntry* oldTable[];

    uint oldSize = _hashSlots;

    if (newSize <= getSlots())
    {
        ret = false;
    }
    else if (ArrayResize(_buckets,newSize) != newSize)
```

```
{
   ret = false;
}
else if (ArrayResize(oldTable,oldSize) != oldSize)
{
   ret = false;
}
else
{
   uint i = 0;
   for(i = 0 ; i < oldSize ; i++ ) oldTable[i] = _buckets[i];
   for(i = 0 ; i<newSize ; i++ ) _buckets[i] = NULL;

   _hashSlots = newSize;
   _resizeThreshold = (int)_hashSlots / 4 * 3;

   for (uint oldHashCode = 0 ; oldHashCode<oldSize ; oldHashCode++)
   {
      OrderGroupHashEntry *next = NULL;

      for (OrderGroupHashEntry *e = oldTable[oldHashCode] ; e != NULL ; e = next)
      {
         next = e._next;

         uint newHashCode = hash(e._key);

         e._next = _buckets[newHashCode];
         _buckets[newHashCode] = e;
      }

      oldTable[oldHashCode] = NULL;
   }
   ret = true;
}
return ret;
```

```
    }

public:
    OrderGroupHashMap()
    {
        init(13, false);
    }

    OrderGroupHashMap(bool adoptValues)
    {
        init(13, adoptValues);
    }

    OrderGroupHashMap(int size)
    {
        init(size, false);
    }

    OrderGroupHashMap(int size, bool adoptValues)
    {
        init(size, adoptValues);
    }

    ~OrderGroupHashMap()
    {
        for(uint i = 0 ; i< _hashSlots ; i++)
        {
            OrderGroupHashEntry *nextEntry = NULL;
            for (OrderGroupHashEntry *entry = _buckets[i] ; entry!= NULL ; entry = nextEntry)
            {
                nextEntry = entry._next;

                if (_adoptValues && entry._val != NULL && CheckPointer(entry._val) == POINTER_DYNAMIC)
                {
                    delete entry._val;
```

```
        }
        delete entry;
      }
      _buckets[i] = NULL;
    }
}

bool ContainsKey(string keyName)
{
   return find(keyName);
}

OrderGroupData* Get(string keyName)
{
   OrderGroupData *obj = NULL;

   if (find(keyName))
   {
      obj = _foundEntry._val;
   }

   return obj;
}

void GetAllData(OrderGroupData* &data[])
{
   for(uint i = 0 ; i < _hashSlots ; i++)
   {
      OrderGroupHashEntry *nextEntry = NULL;
      for (OrderGroupHashEntry *entry = _buckets[i]; entry != NULL ; entry = nextEntry)
      {
         if (entry._val != NULL)
         {
            int size = ArraySize(data);
            ArrayResize(data, size + 1);
```

```
            data[size] = entry._val;
            nextEntry = entry._next;
          }
        }
      }
    }

OrderGroupData* Put(string keyName, OrderGroupData *obj)
{
    OrderGroupData *ret = NULL;

    if (find(keyName))
    {
        ret = _foundEntry._val;

        if (_adoptValues && _foundEntry._val != NULL && CheckPointer(_foundEntry._val) == POINTER_DYNAMIC )
        {
            delete _foundEntry._val;
        }

        _foundEntry._val = obj;

    }
    else
    {
        OrderGroupHashEntry* e = new OrderGroupHashEntry(keyName,obj);
        OrderGroupHashEntry* first = _buckets[_foundIndex];
        e._next = first;
        _buckets[_foundIndex] = e;
        _hashEntryCount++;

        if (_hashEntryCount > _resizeThreshold)
        {
```

```
            rehash(_hashSlots/2*3);
        }
    }
    return ret;
}

bool Delete(string keyName)
{
    bool found = false;

    if (find(keyName))
    {
        OrderGroupHashEntry *next = _foundEntry._next;
        if (_foundPrev != NULL)
        {
            _foundPrev._next = next;
        }
        else
        {
            _buckets[_foundIndex] = next;
        }

        if (_adoptValues && _foundEntry._val != NULL&& CheckPointer(_foundEntry._val) == POINTER_DYNAMIC)
        {
            delete _foundEntry._val;
        }

        delete _foundEntry;
        _hashEntryCount--;
        found=true;
    }
    return found;
}

int DeleteKeys(const string& keys[])
```

```
{
  int count = 0;

  // delete key if found
  for (int i=0; i<ArraySize(keys); i++)
  {
    if (Delete(keys[i]))
      count++;
  }

  return count;
}

int DeleteKeysExcept(const string& keys[])
{
  int index = 0, count = 0;

  string hashedKeys[];
  ArrayResize(hashedKeys, _hashEntryCount);

  for(uint i=0 ; i<_hashSlots ; i++)
  {
    OrderGroupHashEntry *nextEntry = NULL;
    for (OrderGroupHashEntry *entry = _buckets[i] ; entry!= NULL ; entry = nextEntry)
    {
      nextEntry = entry._next;

      if (entry._key != NULL)
      {
        hashedKeys[index] = entry._key;
        index++;
      }
    }
  }
```

```
      // delete other keys if found
      for (int i=0; i<ArraySize(hashedKeys); i++)
      {
        bool keep = false;
        for (int j=0; j<ArraySize(keys); j++)
        {
          if (hashedKeys[i] == keys[j])
          {
            keep = true;
            break;
          }
        }

        if (!keep)
        {
          if (Delete(hashedKeys[i]))
            count++;
        }
      }

      return count;
    }
};

//
//  Wallet
//
class Wallet
{
  private:
    ulong _openedOrderCount;
    ulong _closedOrderCount;

    // Orders currently open
    OrderCollection* _openOrders;
```

```
// Orders currently open by Symbol + Type
OrderGroupHashMap* _openOrdersSymbolType;
// Orders currently open by Symbol
OrderGroupHashMap* _openOrdersSymbol;
// Pending open order
OrderCollection* _pendingOpenOrders;
// Pending close order
OrderCollection* _pendingCloseOrders;
// Most recent closed order
Order* _mostRecentOpenedOrClosedOrder;

OrderRepository* _orderRepository;

void AddOrderToOpenOrderCollections(Order* order)
{
  Order* newOpenOrder = new Order(order);
  _openOrders.Add(newOpenOrder);

  if (IsSymbolOrderTypeOrderGroupActivated())
  {
    string key = GetOrderGroupSymbolOrderTypeKey(order);
    OrderGroupData *orderGroupData = _openOrdersSymbolType.Get(key);
    if (orderGroupData == NULL)
    {
      orderGroupData = new OrderGroupData();
    }

    orderGroupData.OrderGroup.Add(newOpenOrder);

    _openOrdersSymbolType.Put(key, orderGroupData);
  }
  if (IsSymbolOrderGroupActivated())
  {
    string key = GetOrderGroupSymbolKey(order);
    OrderGroupData *orderGroupData = _openOrdersSymbol.Get(key);
```

```
    if (orderGroupData == NULL)
    {
      orderGroupData = new OrderGroupData();
    }

    orderGroupData.OrderGroup.Add(newOpenOrder);

    _openOrdersSymbol.Put(key, orderGroupData);
  }
}

bool RemoveOrderFromOpenOrderCollections(Order* order)
{
  int key = GetOpenOrders().GetKeyByTicket(order.Ticket);
  if (key != -1)
  {
    GetOpenOrders().Remove(key);

    // remove orders from buckets
    if (_openOrdersSymbolType != NULL)
    {
      string symbolOrderTypeKey = GetOrderGroupSymbolOrderTypeKey(order);
      OrderGroupData* openOrdersSymbolTypeData = _openOrdersSymbolType.Get(symbolOrderTypeKey);
      int symbolOrderTypeIndex = openOrdersSymbolTypeData.OrderGroup.GetKeyByTicket(order.Ticket);
      openOrdersSymbolTypeData.OrderGroup.Remove(symbolOrderTypeIndex);
    }
    else if (_openOrdersSymbol != NULL)
    {
      string symbolKey = GetOrderGroupSymbolKey(order);
      OrderGroupData* symbolGroupData = _openOrdersSymbol.Get(symbolKey);
      int symbolIndex = symbolGroupData.OrderGroup.GetKeyByTicket(order.Ticket);
      symbolGroupData.OrderGroup.Remove(symbolIndex);
    }
  }
```

```
    return key != -1;
  }

  string GetOrderGroupSymbolOrderTypeKey(Order* order)
  {
    return order.SymbolCode + IntegerToString(order.Type);
  }

  string GetOrderGroupSymbolKey(Order* order)
  {
    return order.SymbolCode;
  }

  bool IsSymbolOrderTypeOrderGroupActivated()
  {
    return _openOrdersSymbolType != NULL;
  }

  bool IsSymbolOrderGroupActivated()
  {
    return _openOrdersSymbol != NULL;
  }

public:
  void Wallet()
  {
    _openedOrderCount = 0;
    _closedOrderCount = 0;

    _pendingOpenOrders = new OrderCollection();
    _openOrdersSymbolType = NULL;
    _openOrdersSymbol = NULL;
    _pendingCloseOrders = new OrderCollection();
    _orderRepository = new OrderRepository();
```

```
  _openOrders = new OrderCollection();
  _mostRecentOpenedOrClosedOrder = NULL;
}

void ~Wallet()
{
  delete(_pendingOpenOrders);
  delete(_pendingCloseOrders);
  delete(_orderRepository);

  if (_openOrders != NULL)
    delete(_openOrders);

  if (_mostRecentOpenedOrClosedOrder != NULL)
    delete(_mostRecentOpenedOrClosedOrder);

  if (_openOrdersSymbolType != NULL)
    delete(_openOrdersSymbolType);

  if (_openOrdersSymbol != NULL)
    delete (_openOrdersSymbol);
}

void ActivateOrderGroups(ORDER_GROUP_TYPE &groupTypes[])
{
  for (int i = 0; i < ArrayRange(groupTypes,0); i++)
  {
    if (groupTypes[i] == SymbolOrderType && _openOrdersSymbolType == NULL)
    {
      _openOrdersSymbolType = new OrderGroupHashMap();
    }
    else if (groupTypes[i] == SymbolCode && _openOrdersSymbol == NULL)
    {
      _openOrdersSymbol = new OrderGroupHashMap();
    }
```

```
  }
}

OrderCollection* GetOpenOrders()
{
  if (_openOrders == NULL)
    LoadOrdersFromBroker();

  return _openOrders;
}

Order* GetOpenOrder(int ticketId)
{
  int index = _openOrders.GetKeyByTicket(ticketId);
  if (index == -1)
  {
    return NULL;
  }

  return _openOrders.Get(index);
}

void GetOpenOrdersSymbolOrderType(OrderGroupData* &data[])
{
  _openOrdersSymbolType.GetAllData(data);
}

void GetOpenOrdersSymbol(OrderGroupData* &data[])
{
  _openOrdersSymbol.GetAllData(data);
}

OrderCollection* GetPendingOpenOrders()
{
  return _pendingOpenOrders;
```

```
}

OrderCollection* GetPendingCloseOrders()
{
  return _pendingCloseOrders;
}

        void ResetPendingOrders()
{
  delete(_pendingOpenOrders);
  delete(_pendingCloseOrders);

  _pendingOpenOrders = new OrderCollection();
  _pendingCloseOrders = new OrderCollection();

  Print("Wallet has " + IntegerToString(_pendingOpenOrders.Count()) + " pending open orders now.");
  Print("Wallet has " + IntegerToString(_pendingCloseOrders.Count()) + " pending close orders now.");
}

void ResetOpenOrders()
{
  if (_openOrders != NULL)
  {
    delete(_openOrders);
    _openOrders = new OrderCollection();
  }

  if (_openOrdersSymbol != NULL)
  {
    delete(_openOrdersSymbol);
    _openOrdersSymbol = new OrderGroupHashMap();
  }

  if (_openOrdersSymbolType != NULL)
  {
```

```
      delete(_openOrdersSymbolType);
      _openOrdersSymbolType = new OrderGroupHashMap();
    }
  }

  Order* GetLastOpenOrder()
  {
    Order* order = NULL;
    for (int i = _openOrders.Count()-1; i >= 0; i--)
    {
      return _openOrders.Get(i);
    }
    return NULL;
  }

  Order* GetMostRecentOpenedOrClosedOrder()
  {
    return _mostRecentOpenedOrClosedOrder;
  }

  void SetMostRecentOpenedOrClosedOrder(Order* order)
  {
    if (_mostRecentOpenedOrClosedOrder == NULL)
    {
      _mostRecentOpenedOrClosedOrder = new Order(order);
    }
    else if (_mostRecentOpenedOrClosedOrder.CloseTime < order.OpenTime
      || _mostRecentOpenedOrClosedOrder.CloseTime < order.CloseTime
      || (_mostRecentOpenedOrClosedOrder.OpenTime < order.OpenTime &&
_mostRecentOpenedOrClosedOrder.CloseTime == 0))
    {
      delete(_mostRecentOpenedOrClosedOrder);
      _mostRecentOpenedOrClosedOrder = new Order(order);
    }
  }
```

```
void LoadOrdersFromBroker()
{
   OrderCollection* brokerOrders = OrderRepository::GetOpenOrders(MagicNumber);
   for(int i = 0; i < brokerOrders.Count(); i++)
   {
      Order* openOrder = brokerOrders.Get(i);
      AddOrderToOpenOrderCollections(openOrder);

      // Check if order is latest opened or closed order
      SetMostRecentOpenedOrClosedOrder(openOrder);
   }

   // Check if manual closed order is maybe the latest opened or closed order
   Order* lastClosedOrder = OrderRepository::GetLastClosedOrder();
   if (lastClosedOrder != NULL)
   {
    SetMostRecentOpenedOrClosedOrder(lastClosedOrder);
    delete(lastClosedOrder);
   }

   // refactor this later orderCount = _openOrders.Count() didn't work.
   int orderCount = 0;
   if (_openOrders.Count() > 0)
   {
      orderCount = _openOrders.Count();
   }

   delete(brokerOrders);

   Print("Wallet has " + IntegerToString(orderCount) + " orders now.");
}

bool MovePendingOpenToOpenOrders(Order* justOpenedOrder)
{
```

```
  int key = _pendingOpenOrders.GetKeyByTicket(justOpenedOrder.Ticket);
  if (key != -1)
  {
    delete(_mostRecentOpenedOrClosedOrder);
    _mostRecentOpenedOrClosedOrder = new Order(justOpenedOrder);

    AddOrderToOpenOrderCollections(justOpenedOrder);
    delete(justOpenedOrder);

    _pendingOpenOrders.Remove(key);

    _openedOrderCount++;

    return true;
  }

  Alert("Couldn't move pending open order to opened orders for ticketid: " + IntegerToString(justOpenedOrder.Ticket));
  return false;
}

bool CancelPendingOpenOrder(Order* justOpenedOrder)
{
  int key = _pendingOpenOrders.GetKeyByTicket(justOpenedOrder.Ticket);
  if (key != -1)
  {
    delete(justOpenedOrder);
    _pendingOpenOrders.Remove(key);

    return true;
  }

  Alert("Couldn't cancel pending open order for ticketid: " + IntegerToString(justOpenedOrder.Ticket));
  return false;
}
```

```
bool MoveOpenOrderToPendingCloseOrders(Order* orderToClose)
{
  if (RemoveOrderFromOpenOrderCollections(orderToClose))
  {
    _pendingCloseOrders.Add(new Order(orderToClose));

    if (CheckPointer(orderToClose) != POINTER_INVALID
      && CheckPointer(orderToClose) == POINTER_DYNAMIC)
      delete(orderToClose);

    return true;
  }

  Alert("Couldn't move open order to pendingclose orders for ticketid: " + IntegerToString(orderToClose.Ticket));
  return false;
}

bool MovePendingCloseToClosedOrders(Order* justClosedOrder)
{
  int key = _pendingCloseOrders.GetKeyByTicket(justClosedOrder.Ticket);
  if (key != -1)
  {
    delete(_mostRecentOpenedOrClosedOrder);
    _mostRecentOpenedOrClosedOrder = new Order(justClosedOrder);

    _pendingCloseOrders.Remove(key);
    delete(justClosedOrder);

    _closedOrderCount++;

    return true;
  }

  Alert("Couldn't move open order to removed order for ticketid: " + IntegerToString(justClosedOrder.Ticket));
  return false;
```

```
    }

    ulong GetOpenedOrderCount()
    {
      return _openedOrderCount;
    }

    ulong GetClosedOrderCount()
    {
      return _closedOrderCount;
    }
};

//
//  TradeAction
//
enum TradeAction
{
        UnknownAction = 0,
        OpenBuyAction = 1,
        OpenSellAction = 2,
        CloseBuyAction = 3,
        CloseSellAction = 4
};

//
//  AdvisorStrategyExpression interface
//
interface IAdvisorStrategyExpression
{
  bool Evaluate();
};

//
//  TradeSignalCollection
```

```
//
class TradeSignalCollection
{
   private:
      IAdvisorStrategyExpression* _tradeSignals[];
      int _pointer;
      int _size;

   public:
      void TradeSignalCollection()
      {
         _pointer = -1;
         _size = 0;
      }

      void ~TradeSignalCollection()
      {
         for (int i = 0; i < ArraySize(_tradeSignals); i++)
         {
            delete(_tradeSignals[i]);
         }
      }

      void Add(IAdvisorStrategyExpression* item)
      {
         _size = _size + 1;
         ArrayResize(_tradeSignals, _size, 8);

         _tradeSignals[(_size - 1)] = item;
      }

      IAdvisorStrategyExpression* Remove(int index)
      {
         IAdvisorStrategyExpression* removed = NULL;
```

```
   if (index >= 0 && index < _size)
   {
      removed = _tradeSignals[index];

      for (int i = index; i < (_size - 1); i++)
      {
         _tradeSignals[i] = _tradeSignals[i + 1];
      }

      ArrayResize(_tradeSignals, ArraySize(_tradeSignals) - 1, 8);
      _size = _size - 1;
   }

   return removed;
}

IAdvisorStrategyExpression* Get(int index)
{
   if (index >= 0 && index < _size)
   {
      return _tradeSignals[index];
   }

   return NULL;
}

int Count()
{
   return _size;
}

void Rewind()
{
   _pointer = -1;
}
```

```
IAdvisorStrategyExpression* Next()
{
   _pointer++;
   if (_pointer == _size)
   {
      Rewind();
      return NULL;
   }

   return Current();
}

IAdvisorStrategyExpression* Prev()
{
   _pointer--;
   if (_pointer == -1)
   {
      return NULL;
   }

   return Current();
}

bool HasNext()
{
   return (_pointer < (_size - 1));
}

IAdvisorStrategyExpression* Current()
{
   return _tradeSignals[_pointer];
}

int Key()
```

```
    {
        return _pointer;
    }

};

//
//  AdvisorStrategy
//
class AdvisorStrategy
{
        private:
                TradeSignalCollection* _openBuySignals;
                TradeSignalCollection* _openSellSignals;
                TradeSignalCollection* _closeBuySignals;
                TradeSignalCollection* _closeSellSignals;

        public:

                void AdvisorStrategy()
                {
                        _openBuySignals = new TradeSignalCollection();
                        _openSellSignals = new TradeSignalCollection();
                        _closeBuySignals = new TradeSignalCollection();
                        _closeSellSignals = new TradeSignalCollection();
                }

                void ~AdvisorStrategy()
                {
                        delete(_openBuySignals);
                        delete(_openSellSignals);
                        delete(_closeBuySignals);
                        delete(_closeSellSignals);
                }
```

```
bool GetAdvice(TradeAction tradeAction, int level)
{
	if (tradeAction == OpenBuyAction)
	{
		return EvaluateASLevel(_openBuySignals, level);
	}
	else if (tradeAction == OpenSellAction)
	{
		return EvaluateASLevel(_openSellSignals, level);
	}
	else if (tradeAction == CloseBuyAction)
	{
		return EvaluateASLevel(_closeBuySignals, level);
	}
	else if (tradeAction == CloseSellAction)
	{
		return EvaluateASLevel(_closeSellSignals, level);
	}
	else
	{
		Alert("Unsupported OrderType in Advisor Strategy");
	}

	return false;
}

bool EvaluateASLevel(TradeSignalCollection* signals, int level)
{
	if (level > 0
		&& level <= signals.Count())
	{
		return signals.Get(level-1).Evaluate();
	}

	return false;
```

```
}

void RegisterOpenBuy(IAdvisorStrategyExpression* openBuySignal, int level)
{
	if (level <= _openBuySignals.Count())
	{
		Alert("Register Open Buy failed: level already set.");
		return;
	}

	_openBuySignals.Add(openBuySignal);
}

void RegisterOpenSell(IAdvisorStrategyExpression* openSellSignal, int level)
{
	if (level <= _openSellSignals.Count())
	{
		Alert("Register Open Sell failed: level already set.");
		return;
	}

	_openSellSignals.Add(openSellSignal);
}

void RegisterCloseBuy(IAdvisorStrategyExpression* closeBuySignal, int level)
{
	if (level <= _closeBuySignals.Count())
	{
		Alert("Register Close Buy failed: level already set.");
		return;
	}

	_closeBuySignals.Add(closeBuySignal);
}
```

```
		void RegisterCloseSell(IAdvisorStrategyExpression* closeSellSignal, int level)
		{
			if (level <= _closeSellSignals.Count())
			{
				Alert("Register Close Buy failed: level already set.");
				return;
			}

			_closeSellSignals.Add(closeSellSignal);
		}

		int GetNumberOfExpressions(TradeAction tradeAction)
		{
		  if (tradeAction == OpenBuyAction)
			{
				return _openBuySignals.Count();
			}
			else if (tradeAction == OpenSellAction)
			{
				return _openSellSignals.Count();
			}
			else if (tradeAction == CloseBuyAction)
			{
				return _closeBuySignals.Count();
			}
			else if (tradeAction == CloseSellAction)
			{
				return _closeSellSignals.Count();
			}
			return 0;
		}
};

//
//  Indicator Inputs
```

```
//


input int iMA_EMA_M25_period = 25;

input int iMA_EMA_M25_ma_shift = 0;

input int VROC_VROC_PeriodROC = 25;

input int StdDev_StdDevMA_ExtStdDevPeriod = 20;

input int StdDev_StdDevMA_ExtStdDevMAMethod = 0;

input int StdDev_StdDevMA_ExtStdDevAppliedPrice = 0;

input int StdDev_StdDevMA_ExtStdDevShift = 0;


//

//  AdvisorStrategySignals

//


input double OpenBuy_Const_0 = 0.006;


class ASOpenBuyLevel1 : public IAdvisorStrategyExpression

{

  public:


  void ASOpenBuyLevel1()

  {

  }


  bool Evaluate()

  {

    if (((iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,0) >
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))

 &&
((iCustom(Symbol(),PERIOD_H4,"StdDev",StdDev_StdDevMA_ExtStdDevPeriod,StdDev_StdDevMA_ExtStdDevMAMethod,S
tdDev_StdDevMA_ExtStdDevAppliedPrice,StdDev_StdDevMA_ExtStdDevShift,0,0) > OpenBuy_Const_0)

 && (iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,1) <=
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))

)

)

)

    {
```

```
      return true;
    }

    return false;
  }
};


input double OpenSell_Const_0 = 0.006;

class ASOpenSellLevel1 : public IAdvisorStrategyExpression
{
  public:

  void ASOpenSellLevel1()
  {
  }

  bool Evaluate()
  {
    if (((iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,0) <
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
 &&
((iCustom(Symbol(),PERIOD_H4,"StdDev",StdDev_StdDevMA_ExtStdDevPeriod,StdDev_StdDevMA_ExtStdDevMAMethod,S
tdDev_StdDevMA_ExtStdDevAppliedPrice,StdDev_StdDevMA_ExtStdDevShift,0,0) > OpenSell_Const_0)
 && (iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,1) >=
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
)
)
)
    {
      return true;
    }

    return false;
  }
};
```

```
class ASCloseBuyLevel1 : public IAdvisorStrategyExpression
{
  public:

  void ASCloseBuyLevel1()
  {
  }

  bool Evaluate()
  {
    if (((iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,0) <
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
 && (iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,1) >=
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
)
)
    {
      return true;
    }

    return false;
  }
};

class ASCloseSellLevel1 : public IAdvisorStrategyExpression
{
  public:

  void ASCloseSellLevel1()
  {
  }

  bool Evaluate()
  {
```

```
    if (((iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,0) >
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
 && (iCustom(Symbol(),PERIOD_H4,"VROC",VROC_VROC_PeriodROC,0,1) <=
iMA(Symbol(),PERIOD_H4,iMA_EMA_M25_period,iMA_EMA_M25_ma_shift,MODE_EMA,PRICE_CLOSE,0))
)
)
    {
      return true;
    }

    return false;
  }
};

//
//  MoneyManager Interface
//
interface IMoneyManager
{
  double GetLotSize();
  int GetNextLevel(Wallet* wallet);
};

double NormalizeLots(double lots, string pair="")
{
   if (pair == "") pair = Symbol();
   double  lotStep = MarketInfo(pair, MODE_LOTSTEP),
          minLot  = MarketInfo(pair, MODE_MINLOT);
   lots           = MathRound(lots/lotStep) * lotStep;
   if (lots < minLot) lots = minLot;
   return(lots);
}

//
//  MoneyManager Class
//
```

```
input double LotSizePercentageOverride = 0;
double LotSizePercentage = 6;

class MoneyManager : public IMoneyManager
{
  public:
    void MoneyManager()
    {
      if (LotSizePercentageOverride != 0)
      {
        LotSizePercentage = LotSizePercentageOverride;
      }
    }

    double GetLotSize()
    {
      // = [equity/balance] * pippoint * percentage/100.0
      double lotSize = NormalizeLots(NormalizeDouble(AccountBalance() * 0.0001 * LotSizePercentage/100.0, 2));
      return lotSize;
    }

    int GetNextLevel(Wallet* wallet)
    {
      return wallet.GetOpenOrders().Count() + 1;
    }
};

//
//  TradingModuleModuleDemand
//
enum TradingModuleDemand
{
            NoneDemand = 0,
            NoBuyDemand = 1,
            NoSellDemand = 2,
```

```
            NoOpenDemand = 4,
            OpenBuySellDemand = 8,
            OpenBuyDemand = 16,
            OpenSellDemand = 32,
            CloseBuyDemand = 64,
            CloseSellDemand = 128,
            CloseBuySellDemand = 256
};

//
//  TradingModuleExpression Interface
//
interface ITradingModuleSignal
    {
      string GetName();
      bool Evaluate();
    };

interface ITradingModuleValue
    {
      string GetName();
      double Evaluate();
    };

//
//  TradeStrategy Module Interface
//
interface ITradeStrategyModule
    {
      TradingModuleDemand Evaluate(Wallet* wallet, TradingModuleDemand demand, int level = 1);
      void RegisterTradeSignal(ITradingModuleSignal* tradeSignal);
    };

//
//  Open TradeStrategy module Interface
```

```
//
interface ITradeStrategyOpenModule : public ITradeStrategyModule
      {
        TradingModuleDemand EvaluateOpenSignals(Wallet* wallet, TradingModuleDemand demand, int level = 1);
        TradingModuleDemand EvaluateCloseSignals(Wallet* wallet, TradingModuleDemand demand, int level = 1);
      };

//
//  Close TradeStrategy module Interface
//
interface ITradeStrategyCloseModule : public ITradeStrategyModule
      {
        ORDER_GROUP_TYPE GetOrderGroupingType();
        void RegisterTradeValue(ITradingModuleValue* tradeValue);
      };

//
//  TradeStrategy
//
class TradeStrategy
{
   public:
     ITradeStrategyCloseModule* CloseModules[];
   private:

     ITradeStrategyModule* _preventOpenModules[];
     ITradeStrategyOpenModule* _openModule;

     TradingModuleDemand EvaluatePreventOpenModules(Wallet* wallet, TradingModuleDemand preventOpenDemand,
int evaluationLevel = 1)
     {
       // Check PreventOpen demands
       TradingModuleDemand preventOpenDemands[];
       ArrayResize(preventOpenDemands, ArraySize(_preventOpenModules), 8);
```

```
    for (int i = 0; i < ArraySize(_preventOpenModules); i++)
    {
      preventOpenDemands[i] = _preventOpenModules[i].Evaluate(wallet, NoneDemand, evaluationLevel);
    }

    // Combine the advices into 1 advice (None, NoBuy, NoSell or NoOpen)
    return PreventOpenModuleBase::GetCombinedPreventOpenDemand(preventOpenDemands);
  }

  TradingModuleDemand EvaluateCloseModules(Wallet* wallet, TradingModuleDemand closeDemand, int evaluationLevel = 1)
  {
    // Check Close demands
    TradingModuleDemand closeDemands[];
    ArrayResize(closeDemands, ArraySize(CloseModules), 8);

    for (int i = 0; i < ArraySize(CloseModules); i++)
    {
      closeDemands[i] = CloseModules[i].Evaluate(wallet, NoneDemand, evaluationLevel);
    }

    // Combine the advices into 1 advice (None, NoBuy, NoSell or NoOpen)
    return CloseModuleBase::GetCombinedCloseDemand(closeDemands);
  }

  // This method puts orders in the pendingCloseOrders collection if the TP/ SL are hit
  void EvaluateCloseConditions(Wallet* wallet, TradingModuleDemand signalDemand)
  {
    OrderCollection* openOrders = wallet.GetOpenOrders();
    if (openOrders.Count() == 0)
    {
      return;
    }

    // First open order
```

```
int orderTypeOfOpeningOrder = wallet.GetOpenOrders().Get(0).Type;

for (int i = openOrders.Count()-1; i >= 0; i--)
{
  Order* order = openOrders.Get(i);
  double stopLossHit = (order.StopLoss != 0 &&
    (  (order.Type == OP_BUY && Bid <= order.StopLoss) ||
      (order.Type == OP_SELL && Ask >= order.StopLoss)));
  double takeProfitHit = (order.TakeProfit != 0 &&
    (  (order.Type == OP_BUY && Bid >= order.TakeProfit) ||
      (order.Type == OP_SELL && Ask <= order.TakeProfit)));
                                double closeSignal =
    (order.Type == OP_BUY && signalDemand == CloseBuyDemand)
    || (order.Type == OP_SELL && signalDemand == CloseSellDemand)
    || signalDemand == CloseBuySellDemand;

  if (stopLossHit || takeProfitHit || closeSignal)
  {
    // Evaluate prevent open modules, use level 0 so the modules know it's used for evaluating closing orders
    TradingModuleDemand finalPreventOpenAdvice = EvaluatePreventOpenModules(wallet, NoneDemand, 0);

    // Evaluate Open module as if there are no open orders and inform module of NoOpen demands
    TradingModuleDemand openDemand = _openModule.EvaluateOpenSignals(wallet, finalPreventOpenAdvice, 1);

    if ((orderTypeOfOpeningOrder == OP_BUY && openDemand == OpenBuyDemand)
      || (orderTypeOfOpeningOrder == OP_SELL && openDemand == OpenSellDemand)
      || (openDemand == OpenBuySellDemand))
      {
        // block close, because the order will be opened straight away again anyway
        return;
      }

    // Move order to PendingCloseOrders
    wallet.MoveOpenOrderToPendingCloseOrders(order);
  }
```

```
    }
  }

public:
  void TradeStrategy(ITradeStrategyOpenModule* openModule)
  {
    _openModule = openModule;
  }

  void ~TradeStrategy()
  {
    for (int i=ArraySize(_preventOpenModules)-1; i >= 0; i--)
    {
      delete(_preventOpenModules[i]);
    }

    delete(_openModule);

    for (int i=ArraySize(CloseModules)-1; i >= 0; i--)
    {
      delete(CloseModules[i]);
    }
  }

  void Evaluate(Wallet* wallet)
  {
    int orderCount = wallet.GetOpenOrders().Count();

    TradingModuleDemand finalPreventOpenAdvice = EvaluatePreventOpenModules(wallet, NoneDemand, orderCount +
1);

    if (orderCount > 0)
    {
      // Close modules set the TP/ SL in memory, don't persist to broker (optionally) or their dealing desk will screw you
over
      EvaluateCloseModules(wallet, NoneDemand);
```

```
      TradingModuleDemand signalDemand = _openModule.EvaluateCloseSignals(wallet, finalPreventOpenAdvice, 0);

      // TP and SL can be modified by multiple modules. Here we evaluate the order's TP/SL with the current quote
      EvaluateCloseConditions(wallet, signalDemand);
    }

    // Evaluate Open module and inform module of NoOpen demands
    _openModule.Evaluate(wallet, finalPreventOpenAdvice, 0);
  }

  void RegisterPreventOpenModule(ITradeStrategyModule* preventOpenModule)
  {
    int size = ArraySize(_preventOpenModules);
    ArrayResize(_preventOpenModules, size + 1, 8);
    _preventOpenModules[size] = preventOpenModule;
  }

  void RegisterCloseModule(ITradeStrategyCloseModule* closeModule)
  {
    int size = ArraySize(CloseModules);
    ArrayResize(CloseModules, size + 1, 8);
    CloseModules[size] = closeModule;
  }
};

//
//  OpenModule BaseClass
//
class OpenModuleBase : public ITradeStrategyOpenModule
{
 protected:
  AdvisorStrategy* _advisorStrategy;
  IMoneyManager* _moneyManager;
```

```
  Order* OpenOrder(int orderType)
  {
    Order* order = new Order();
    order.Type = orderType;
    order.MagicNumber = MagicNumber;
    order.Lots = _moneyManager.GetLotSize();
    if (order.Type == OP_BUY)
    {
      order.OpenPrice = Ask;
    }
    else if (order.Type == OP_SELL)
    {
      order.OpenPrice = Bid;
    }
    order.SymbolCode = Symbol();
                        order.Comment = stopLossComment + takeProfitComment;
    order.LowestProfitPips = DBL_MAX;
    order.HighestProfitPips = -DBL_MAX;

    return order;
  }

public:
  void OpenModuleBase(AdvisorStrategy* advisorStrategy, IMoneyManager* moneyManager)
  {
    _advisorStrategy = advisorStrategy;
    _moneyManager = moneyManager;
  }

  void GetTradeActions(Wallet* wallet, TradingModuleDemand preventOpenDemand, TradeAction& result[])
  {
    TradeAction tempresult[];

    if (wallet.GetOpenOrders().Count() > 0)
    {
```

```
  // First open order is buy? then only reevaluate the buy AS levels for adding buys
  Order* firstOrder = wallet.GetOpenOrders().Get(0);
  if (firstOrder.Type == OP_BUY)
  {
    ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
    tempresult[0] = OpenBuyAction;

                                ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
    tempresult[1] = CloseBuyAction;
  }
  else if (firstOrder.Type == OP_SELL)
  {
    ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
    tempresult[0] = OpenSellAction;

                                ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
    tempresult[1] = CloseSellAction;
  }
  else
  {
    Alert("Unsupported ordertype");
  }
}
else
{
  ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
  tempresult[0] = OpenBuyAction;

  ArrayResize(tempresult, ArraySize(tempresult) + 1, 8);
  tempresult[1] = OpenSellAction;
}

// filter prevent open demands
for(int i = 0; i < ArraySize(tempresult); i++)
{
```

```
    if (preventOpenDemand == NoOpenDemand
        || (preventOpenDemand == NoBuyDemand && tempresult[i] == OpenBuyAction)
        || (preventOpenDemand == NoSellDemand && tempresult[i] == OpenSellAction))
      {
        continue;
      }

    ArrayResize(result, ArraySize(result) + 1, 8);
    result[ArraySize(result)-1] = tempresult[i];
  }
}

virtual void RegisterTradeSignal(ITradingModuleSignal* tradeSignal)
{
        // nothing to do here
}

static TradingModuleDemand GetCombinedOpenDemand(TradingModuleDemand &openDemands[])
{
  TradingModuleDemand result = NoneDemand;

  for (int i = 0; i < ArraySize(openDemands); i++)
  {
    if (result == OpenBuySellDemand)
      return OpenBuySellDemand;

    if (openDemands[i] == OpenBuySellDemand)
    {
      result = OpenBuySellDemand;
    }
    else if (result == NoneDemand && openDemands[i] == OpenBuyDemand)
    {
      result = OpenBuyDemand;
    }
    else if (result == NoneDemand && openDemands[i] == OpenSellDemand)
```

```
    {
      result = OpenSellDemand;
    }
    else if (result == OpenBuyDemand && openDemands[i] == OpenSellDemand)
    {
      result = OpenBuySellDemand;
    }
    else if (result == OpenSellDemand && openDemands[i] == OpenBuyDemand)
    {
      result = OpenBuySellDemand;
    }
  }
  return result;
}

        static TradingModuleDemand GetCombinedCloseDemand(TradingModuleDemand &closeDemands[])
{
  TradingModuleDemand result = NoneDemand;

  for (int i = 0; i < ArraySize(closeDemands); i++)
  {
    if (result == CloseBuySellDemand)
      return CloseBuySellDemand;

    if (closeDemands[i] == CloseBuySellDemand)
    {
      result = CloseBuySellDemand;
    }
    else if (result == NoneDemand && closeDemands[i] == CloseBuyDemand)
    {
      result = CloseBuyDemand;
    }
    else if (result == NoneDemand && closeDemands[i] == CloseSellDemand)
    {
      result = CloseSellDemand;
```

```
        }
        else if (result == CloseBuyDemand && closeDemands[i] == CloseSellDemand)
        {
          result = CloseBuySellDemand;
        }
        else if (result == CloseSellDemand && closeDemands[i] == CloseBuyDemand)
        {
          result = CloseBuySellDemand;
        }
      }
      return result;
    }

    int GetNumberOfOpenOrders(Wallet* wallet)
    {
      return wallet.GetOpenOrders().Count();
    }
};

//
//  Single-order OpenModule 1
//
class SingleOpenModule_1 : public OpenModuleBase
{
  public:

    void SingleOpenModule_1(AdvisorStrategy* advisorStrategy, IMoneyManager* moneyManager)
      : OpenModuleBase(advisorStrategy, moneyManager)
    {
    }

    TradingModuleDemand Evaluate(Wallet* wallet, TradingModuleDemand preventOpenDemand, int level)
    {
      TradingModuleDemand signalsDemand = EvaluateSignals(wallet, preventOpenDemand, level);
      if (signalsDemand != NoneDemand)
```

```
    {
      EvaluateOpenConditions(wallet, signalsDemand);
    }
    return signalsDemand;
  }

  TradingModuleDemand EvaluateOpenSignals(Wallet* wallet, TradingModuleDemand preventOpenDemand, int
requestedEvaluationLevel)
  {
    TradingModuleDemand openDemands[];

    // Get adviced open ordertypes
    TradeAction tradeActionsToEvaluate[];
    GetTradeActions(wallet, preventOpenDemand, tradeActionsToEvaluate);

    int moneyManagementLevel;
    if (requestedEvaluationLevel == 0)
    {
      moneyManagementLevel = _moneyManager.GetNextLevel(wallet);
    }
    else
    {
      moneyManagementLevel = requestedEvaluationLevel;
    }

    for(int i = 0; i < ArraySize(tradeActionsToEvaluate); i++)
    {
      int tradeActionEvaluationLevel;
      if (tradeActionsToEvaluate[i] == CloseBuyAction
          || tradeActionsToEvaluate[i] == CloseSellAction)
      {
        continue;
      }
      else
      {
```

```
            tradeActionEvaluationLevel = moneyManagementLevel;
          }

          if (_advisorStrategy.GetAdvice(tradeActionsToEvaluate[i], tradeActionEvaluationLevel))
          {
            if (tradeActionsToEvaluate[i] == OpenBuyAction)
            {
              int size = ArraySize(openDemands);
              int newSize = size + 1;
              ArrayResize(openDemands, newSize, 8);
              openDemands[newSize-1] = OpenBuyDemand;
            }
            else if (tradeActionsToEvaluate[i] == OpenSellAction)
            {
              int size = ArraySize(openDemands);
              int newSize = size + 1;
              ArrayResize(openDemands, newSize, 8);
              openDemands[newSize-1] = OpenSellDemand;
            }
          }
        }

        TradingModuleDemand combinedOpenSignalDemand =
OpenModuleBase::GetCombinedOpenDemand(openDemands);
        return combinedOpenSignalDemand;
    }

    TradingModuleDemand EvaluateCloseSignals(Wallet* wallet, TradingModuleDemand preventOpenDemand, int
requestedEvaluationLevel)
    {
        TradingModuleDemand closeDemands[];
        // Get adviced open ordertypes
        TradeAction tradeActionsToEvaluate[];
        GetTradeActions(wallet, preventOpenDemand, tradeActionsToEvaluate);

        int moneyManagementLevel;
```

```
if (requestedEvaluationLevel == 0)
{
  moneyManagementLevel = _moneyManager.GetNextLevel(wallet);
}
else
{
  moneyManagementLevel = requestedEvaluationLevel;
}

for(int i = 0; i < ArraySize(tradeActionsToEvaluate); i++)
{
  int tradeActionEvaluationLevel;
  if (tradeActionsToEvaluate[i] == CloseBuyAction
     || tradeActionsToEvaluate[i] == CloseSellAction)
  {
    tradeActionEvaluationLevel = moneyManagementLevel - 1;
  }
  else
  {
    continue;
  }

  if (_advisorStrategy.GetAdvice(tradeActionsToEvaluate[i], tradeActionEvaluationLevel))
  {
    if (tradeActionsToEvaluate[i] == CloseBuyAction)
    {
      int size = ArraySize(closeDemands);
      int newSize = size + 1;
      ArrayResize(closeDemands, newSize, 8);
      closeDemands[newSize - 1] = CloseBuyDemand;
    }
    else if (tradeActionsToEvaluate[i] == CloseSellAction)
    {
      int size = ArraySize(closeDemands);
      int newSize = size + 1;
```

```
            ArrayResize(closeDemands, newSize, 8);
            closeDemands[newSize - 1] = CloseSellDemand;
          }
        }
      }
      TradingModuleDemand combinedCloseSignalDemand =
OpenModuleBase::GetCombinedCloseDemand(closeDemands);
      return combinedCloseSignalDemand;
    }

  private:
    TradingModuleDemand EvaluateSignals(Wallet* wallet, TradingModuleDemand preventOpenDemand, int level)
    {
      TradingModuleDemand openDemands[];
      TradingModuleDemand closeDemands[];

      // Get adviced open ordertypes
      TradeAction tradeActionsToEvaluate[];
      GetTradeActions(wallet, preventOpenDemand, tradeActionsToEvaluate);

      if (level == 0)
      {
        level = _moneyManager.GetNextLevel(wallet);
      }

      for(int i = 0; i < ArraySize(tradeActionsToEvaluate); i++)
      {
        // Single order module will never evaluate level 2, however it should evaluate level 1 close signals
        if (tradeActionsToEvaluate[i] == CloseBuyAction
            || tradeActionsToEvaluate[i] == CloseSellAction)
        {
          level = level - 1;
        }

        if (_advisorStrategy.GetAdvice(tradeActionsToEvaluate[i], level))
```

```
    {
      if (tradeActionsToEvaluate[i] == OpenBuyAction)
      {
        int size = ArraySize(openDemands);
        int newSize = size + 1;
        ArrayResize(openDemands, newSize, 8);
        openDemands[newSize-1] = OpenBuyDemand;
      }
      else if (tradeActionsToEvaluate[i] == OpenSellAction)
      {
        int size = ArraySize(openDemands);
        int newSize = size + 1;
        ArrayResize(openDemands, newSize, 8);
        openDemands[newSize-1] = OpenSellDemand;
      }
      else if (tradeActionsToEvaluate[i] == CloseBuyAction)
      {
        int size = ArraySize(closeDemands);
        int newSize = size + 1;
        ArrayResize(closeDemands, newSize, 8);
        closeDemands[newSize - 1] = CloseBuyDemand;
      }
      else if (tradeActionsToEvaluate[i] == CloseSellAction)
      {
        int size = ArraySize(closeDemands);
        int newSize = size + 1;
        ArrayResize(closeDemands, newSize, 8);
        closeDemands[newSize - 1] = CloseSellDemand;
      }
    }
  }

  TradingModuleDemand combinedCloseSignalDemand =
OpenModuleBase::GetCombinedCloseDemand(closeDemands);
  if (combinedCloseSignalDemand != NoneDemand)
```

```
      {
        return combinedCloseSignalDemand;
      }

      TradingModuleDemand combinedOpenSignalDemand =
OpenModuleBase::GetCombinedOpenDemand(openDemands);
      return combinedOpenSignalDemand;
    }

    // This method puts orders in the pendingOpenOrders collection if the advisor says it should open orders
    void EvaluateOpenConditions(Wallet* wallet, TradingModuleDemand signalDemand)
    {
      // Open orders based on openDemand
      if (signalDemand == OpenBuyDemand)
      {
        wallet.GetPendingOpenOrders().Add(OpenOrder(OP_BUY));
      }
      else if (signalDemand == OpenSellDemand)
      {
        wallet.GetPendingOpenOrders().Add(OpenOrder(OP_SELL));
      }
      else if (signalDemand == OpenBuySellDemand)
      {
        wallet.GetPendingOpenOrders().Add(OpenOrder(OP_BUY));
        wallet.GetPendingOpenOrders().Add(OpenOrder(OP_SELL));
      }
    }
};

//
//  CloseModule BaseClass
//
class CloseModuleBase : public ITradeStrategyCloseModule
{
  protected:
```

```
bool SetNewTakeProfit(Order* order, double possibleNewTakeProfit)
{
  return SetNewTakeProfit(order, possibleNewTakeProfit, "");
}

bool SetNewTakeProfit(Order* order, double possibleNewTakeProfit, string comment)
{
  double newTakeProfit = order.TakeProfit == NULL ? possibleNewTakeProfit : order.Type == OP_BUY ? MathMin((double)order.TakeProfit, possibleNewTakeProfit) : MathMax(order.TakeProfit, possibleNewTakeProfit);
  if (order.TakeProfit == NULL || MathAbs(newTakeProfit - (double)order.TakeProfit) > 0)
  {
                                takeProfitComment = comment; // global used when opening new order (since that's the only moment where you can put a comment)

    order.TakeProfit = newTakeProfit;
    return true;
  }
  return false;
}

bool SetNewStopLoss(Order* order, double possibleNewStopLoss)
{
  return SetNewStopLoss(order, possibleNewStopLoss, "");
}

bool SetNewStopLoss(Order* order, double possibleNewStopLoss, string comment)
{
  double newStopLoss = order.StopLoss == NULL ? possibleNewStopLoss : order.Type == OP_BUY ? MathMax((double)order.StopLoss, possibleNewStopLoss) : MathMin(order.StopLoss, possibleNewStopLoss);
  if (order.StopLoss == NULL || MathAbs(newStopLoss - order.StopLoss) > 0)
  {
    stopLossComment = comment; // global used when opening new order (since that's the only moment where you can put a comment)

    //Print("New stop loss set.");
    order.StopLoss = newStopLoss;
    return true;
```

```
    }
    return false;
  }

  void SetOrderStopLossToClosePrice(Order* openOrder, string moduleComment = "")
  {
    double possibleNewStopLoss = GetClosePrice(openOrder.Type);
    if (SetNewStopLoss(openOrder, possibleNewStopLoss))
    {
      //Print("SL set: " + possibleNewStopLoss);
      if (moduleComment != "")
      {
        openOrder.Comment = moduleComment; //"Order closed by StopLossCloseModule";
      }
    }
  }

  void SetOrderTakeProfitToClosePrice(Order* openOrder, string moduleComment = "")
  {
    double possibleNewTakeProfit = CloseModuleBase::GetClosePrice(openOrder.Type);
    if (SetNewTakeProfit(openOrder, possibleNewTakeProfit))
    {
      if (moduleComment != "")
      {
        openOrder.Comment = moduleComment;
      }
    }
  }

  double CalculateOrderProfitSingleOrder(Order* order, ORDER_GROUP_TYPE groupType,
ORDER_PROFIT_CALCULATION_TYPE calculationType)
  {
    if (calculationType == Pips)
    {
      return order.CalculateProfitPips();
```

```
    }
    else if (calculationType == Money)
    {
      return order.CalculateProfitCurrency();
    }
    else if (calculationType == EquityPercentage)
    {
      return order.CalculateProfitEquityPercentage();
    }
    else
    {
      Alert("Can't execute CalculateOrderProfit. Unknown calculationType: " + IntegerToString(calculationType) );
    }

    return 0;
  }

  double CalculateOrderCollectionProfit(OrderCollection &openOrders, ORDER_PROFIT_CALCULATION_TYPE
calculationType)
  {
    double collectionProfit = 0;
    for(int i = 0; i < openOrders.Count(); i++)
    {
      Order* openOrder = openOrders.Get(i);
      if (calculationType == Pips)
      {
        collectionProfit += openOrder.CalculateProfitPips();
      }
      else if (calculationType == Money)
      {
        collectionProfit += openOrder.CalculateProfitCurrency();
      }
      else if (calculationType == EquityPercentage)
      {
        collectionProfit += openOrder.CalculateProfitEquityPercentage();
```

```
        }
        else
        {
          Alert("Can't execute CalculateOrderCollectionProfit. Unknown calculationType: " +
IntegerToString(calculationType) );
        }
      }

      return collectionProfit;
    }

    static double GetClosePrice(int orderType)
    {
      switch (orderType)
      {
        case OP_SELL:
          return Ask;
        case OP_BUY:
          return Bid;
        default:
          return 0;
      }
    }

public:
    virtual void RegisterTradeSignal(ITradingModuleSignal* tradeSignal)
    {
            // nothing to do here
    }

    virtual void RegisterTradeValue(ITradingModuleValue* tradeValue)
    {
            // nothing to do here
    }
```

```
    static TradingModuleDemand GetCombinedCloseDemand(TradingModuleDemand &closeDemands[])
    {
      TradingModuleDemand result = NoneDemand;

      for (int i = 0; i < ArraySize(closeDemands); i++)
                        {
        if (result == CloseBuySellDemand)
          return CloseBuySellDemand;

        if (closeDemands[i] == CloseBuySellDemand)
        {
          result = CloseBuySellDemand;
        }
        else if (result == NoneDemand && closeDemands[i] == CloseBuyDemand)
        {
          result = CloseBuyDemand;
        }
        else if (result == NoneDemand && closeDemands[i] == CloseSellDemand)
        {
          result = CloseSellDemand;
        }
        else if (result == CloseBuyDemand && closeDemands[i] == CloseSellDemand)
        {
          result = CloseBuySellDemand;
        }
        else if (result == CloseSellDemand && closeDemands[i] == CloseBuyDemand)
        {
          result = CloseBuySellDemand;
        }
      }
      return result;
    }

    virtual ORDER_GROUP_TYPE GetOrderGroupingType() = NULL;
};
```

```
//
//  StopLoss CloseModule 1
//

input double StopLossModuleValue1 = 0.5;
ORDER_GROUP_TYPE StopLossModuleGroupType1 = Single;
ORDER_PROFIT_CALCULATION_TYPE StopLossModuleProfitCalculationType1 = EquityPercentage;

class StopLossCloseModule_1 : public CloseModuleBase
{
  public:

    void StopLossCloseModule_1()
    {
    }

    TradingModuleDemand Evaluate(Wallet* wallet, TradingModuleDemand demand, int level = 1)
    {
      if (StopLossModuleGroupType1 == Single)
      {
        OrderCollection* openOrders = wallet.GetOpenOrders();
        for(int i = 0; i < openOrders.Count(); i++)
        {
          Order* openOrder = openOrders.Get(i);
          double profit = CalculateOrderProfitSingleOrder(openOrder, StopLossModuleGroupType1,
StopLossModuleProfitCalculationType1);
          if (profit <= -StopLossModuleValue1)
          {
            SetOrderStopLossToClosePrice(openOrder, "Order SL set by StopLossCloseModule");
          }
        }
      }
      else if (StopLossModuleGroupType1 == SymbolOrderType)
      {
        OrderGroupData* dataSymbolOrderType[];
```

```
      wallet.GetOpenOrdersSymbolOrderType(dataSymbolOrderType);


      for(int i = 0; i < ArraySize(dataSymbolOrderType); i++)
      {
        double profit = CalculateOrderCollectionProfit(dataSymbolOrderType[i].OrderGroup,
StopLossModuleProfitCalculationType1);
        if (profit <= -StopLossModuleValue1)
        {
          for(int j = 0; j < dataSymbolOrderType[i].OrderGroup.Count(); j++)
          {
            Order* openOrder = dataSymbolOrderType[i].OrderGroup.Get(j);


            SetOrderStopLossToClosePrice(openOrder, "Order SL set by StopLossCloseModule");
          }
        }
      }
    }
    else if (StopLossModuleGroupType1 == SymbolCode)
    {
      OrderGroupData* dataSymbol[];
      wallet.GetOpenOrdersSymbol(dataSymbol);


      for(int i = 0; i < ArraySize(dataSymbol); i++)
      {
        double profit = CalculateOrderCollectionProfit(dataSymbol[i].OrderGroup,
StopLossModuleProfitCalculationType1);
        if (profit <= -StopLossModuleValue1)
        {
          for(int j = 0; j < dataSymbol[i].OrderGroup.Count(); j++)
          {
            Order* openOrder = dataSymbol[i].OrderGroup.Get(j);
            SetOrderStopLossToClosePrice(openOrder, "Order SL set by StopLossCloseModule");
          }
        }
      }
    }
```

```
  else if (StopLossModuleGroupType1 == Basket)
  {
     OrderCollection* currentOpenOrders = OrderRepository::GetOpenOrders(MagicNumber, NULL);

    if (currentOpenOrders.Count() > 0)
    {
      double profit = CalculateOrderCollectionProfit(currentOpenOrders, StopLossModuleProfitCalculationType1);
      if (profit <= -StopLossModuleValue1)
      {
        for(int i = 0; i < currentOpenOrders.Count(); i++)
        {
          Order* openOrder = currentOpenOrders.Get(i);
          Order* thisEaOrder = wallet.GetOpenOrder(openOrder.Ticket);
          if (thisEaOrder != NULL)
          {
            // order/ position from this EA
            SetOrderStopLossToClosePrice(thisEaOrder, "Order SL set by StopLossCloseModule");
          }
          else
          {
            // order/ position from another EA
            OrderRepository::CloseOrder(openOrder);
          }
        }
      }
    }
  }

  return NoneDemand;
}

ORDER_GROUP_TYPE GetOrderGroupingType()
{
  return StopLossModuleGroupType1;
}
```

```
};

//
//  TakeProfit CloseModule 1
//

input double TakeProfitModuleValue1 = 1;
ORDER_GROUP_TYPE TakeProfitModuleGroupType1 = Single;
ORDER_PROFIT_CALCULATION_TYPE TakeProfitModuleProfitCalculationType1 = EquityPercentage;

class TakeProfitCloseModule_1 : public CloseModuleBase
{
  public:

    void TakeProfitCloseModule_1()
    {
    }

    TradingModuleDemand Evaluate(Wallet* wallet, TradingModuleDemand demand, int level = 1)
    {
      if (TakeProfitModuleGroupType1 == Single)
      {
        OrderCollection* openOrders = wallet.GetOpenOrders();
        for(int i = 0; i < openOrders.Count(); i++)
        {
          Order* openOrder = openOrders.Get(i);

           double profit = CalculateOrderProfitSingleOrder(openOrder, TakeProfitModuleGroupType1,
TakeProfitModuleProfitCalculationType1);
           if (profit >= TakeProfitModuleValue1)
           {
             SetOrderTakeProfitToClosePrice(openOrder, "Order TP set by TakeProfitCloseModule");
           }
        }
      }
```

```
    else if (TakeProfitModuleGroupType1 == SymbolOrderType)
    {
      OrderGroupData* dataSymbolOrderType[];
      wallet.GetOpenOrdersSymbolOrderType(dataSymbolOrderType);

      for(int i = 0; i < ArraySize(dataSymbolOrderType); i++)
      {
        double profit = CalculateOrderCollectionProfit(dataSymbolOrderType[i].OrderGroup,
TakeProfitModuleProfitCalculationType1);
        if (profit >= TakeProfitModuleValue1)
        {
          for(int j = 0; j < dataSymbolOrderType[i].OrderGroup.Count(); j++)
          {
            Order* openOrder = dataSymbolOrderType[i].OrderGroup.Get(j);

            SetOrderTakeProfitToClosePrice(openOrder, "Order TP set by TakeProfitCloseModule");
          }
        }
      }
    }
    else if (TakeProfitModuleGroupType1 == SymbolCode)
    {
      OrderGroupData* dataSymbol[];
      wallet.GetOpenOrdersSymbol(dataSymbol);

      for(int i = 0; i < ArraySize(dataSymbol); i++)
      {
        double profit = CalculateOrderCollectionProfit(dataSymbol[i].OrderGroup,
TakeProfitModuleProfitCalculationType1);
        if (profit >= TakeProfitModuleValue1)
        {
          for(int j = 0; j < dataSymbol[i].OrderGroup.Count(); j++)
          {
            Order* openOrder = dataSymbol[i].OrderGroup.Get(j);
            SetOrderTakeProfitToClosePrice(openOrder, "Order TP set by TakeProfitCloseModule");
          }
```

```
      }
    }
  }
  else if (TakeProfitModuleGroupType1 == Basket)
  {
    OrderCollection* currentOpenOrders = OrderRepository::GetOpenOrders(MagicNumber, NULL);

    if (currentOpenOrders.Count() > 0)
    {
      double profit = CalculateOrderCollectionProfit(currentOpenOrders, TakeProfitModuleProfitCalculationType1);
      if (profit >= TakeProfitModuleValue1)
      {
        for(int i = 0; i < currentOpenOrders.Count(); i++)
        {
          Order* openOrder = currentOpenOrders.Get(i);
          Order* thisEaOrder = wallet.GetOpenOrder(openOrder.Ticket);
          if (thisEaOrder != NULL)
          {
            // order/ position from this EA
            SetOrderTakeProfitToClosePrice(thisEaOrder, "Order SL set by TakeProfitCloseModule");
          }
          else
          {
            // order/ position from another EA
            OrderRepository::CloseOrder(openOrder);
          }
        }
      }
    }
  }

  return NoneDemand;
}

ORDER_GROUP_TYPE GetOrderGroupingType()
```

```
    {
      return TakeProfitModuleGroupType1;
    }
};

//
//  PreventOpenModule BaseClass
//
class PreventOpenModuleBase : public ITradeStrategyModule
{
  public:
    virtual void RegisterTradeSignal(ITradingModuleSignal* tradeSignal)
    {

    }

    static TradingModuleDemand GetCombinedPreventOpenDemand(TradingModuleDemand &preventOpenAdvices[])
    {
      TradingModuleDemand result = NoneDemand;

      for (int i = 0; i < ArraySize(preventOpenAdvices); i++)
      {
        if (result == NoOpenDemand)
          return NoOpenDemand;

        if (preventOpenAdvices[i] == NoOpenDemand)
        {
          result = NoOpenDemand;
        }
        else if (result == NoneDemand && preventOpenAdvices[i] == NoBuyDemand)
        {
          result = NoBuyDemand;
        }
        else if (result == NoneDemand && preventOpenAdvices[i] == NoSellDemand)
        {
```

```
            result = NoSellDemand;
          }
          else if (result == NoBuyDemand && preventOpenAdvices[i] == NoSellDemand)
          {
            result = NoOpenDemand;
          }
          else if (result == NoSellDemand && preventOpenAdvices[i] == NoBuyDemand)
          {
            result = NoOpenDemand;
          }
        }
        return result;
      }
};

//
//  Trader Interface
//
interface ITrader
{
  void HandleTick();
  void Init();
};

ITrader *_ea;

//
//  Expert Advisor Class
//
class EA : public ITrader
{
   private:
      bool _firstTick;
      TradeStrategy* _tradeStrategy;
      AdvisorStrategy* _advisorStrategy;
```

```
   IMoneyManager* _moneyManager;
   Wallet* _wallet;

public:
   void EA()
   {
      _firstTick = true;

      _wallet = new Wallet();
      _wallet.LoadOrdersFromBroker();

      // Advisor Strategy
      _advisorStrategy = new AdvisorStrategy();
      _advisorStrategy.RegisterOpenBuy(new ASOpenBuyLevel1(), 1);
      _advisorStrategy.RegisterOpenSell(new ASOpenSellLevel1(), 1);
      _advisorStrategy.RegisterCloseBuy(new ASCloseBuyLevel1(), 1);
      _advisorStrategy.RegisterCloseSell(new ASCloseSellLevel1(), 1);

      // MoneyManager
      _moneyManager = new MoneyManager();

      // Trader Strategy
      _tradeStrategy = new TradeStrategy(new SingleOpenModule_1(_advisorStrategy, _moneyManager));

      // Dynamic Modules
      _tradeStrategy.RegisterCloseModule(new StopLossCloseModule_1());
      _tradeStrategy.RegisterCloseModule(new TakeProfitCloseModule_1());


   }

   void ~EA()
   {
      delete(_tradeStrategy);
      delete(_moneyManager);
```

```
   delete(_advisorStrategy);
   delete(_wallet);

}

void Init()
{
   SetOrderGrouping();
}

void HandleTick()
{
   if (StopEA)
   {
      return;
   }

   // Only check on live trading
   if (!IsTesting())
   {
     // if number of open orders is incorrect, reset in memory pending orders and load open orders
     SyncOrders();
   }


   // Update orders with latest tick quote
   UpdateOrders();

   if (_wallet.GetPendingOpenOrders().Count() == 0 && _wallet.GetPendingCloseOrders().Count() == 0)
   {
     _tradeStrategy.Evaluate(_wallet);
   }

   if (ExecutePendingCloseOrders())
   {
```

```
      ExecutePendingOpenOrders();
    }

    if (_firstTick)
      _firstTick = false;
  }

  private:

  void SetOrderGrouping()
  {
    int size = ArraySize(_tradeStrategy.CloseModules);
    ORDER_GROUP_TYPE groups[];
    ArrayResize(groups, size);

    for(int i = 0; i < ArraySize(_tradeStrategy.CloseModules); i++)
    {
      groups[i] = _tradeStrategy.CloseModules[i].GetOrderGroupingType();
    }

    _wallet.ActivateOrderGroups(groups);
  }

  void SyncOrders()
  {
    OrderCollection* currentOpenOrders = OrderRepository::GetOpenOrders(MagicNumber, NULL);
    if (currentOpenOrders.Count() != _wallet.GetOpenOrders().Count())
    {
      Print("Manual orderchanges detected" + " (found in MT: " + IntegerToString(currentOpenOrders.Count()) + " and in wallet: " + IntegerToString(_wallet.GetOpenOrders().Count()) + "), resetting EA, loading open orders.");

      // An order was manually opened or closed, we reset everything
      _wallet.ResetOpenOrders();
      _wallet.ResetPendingOrders();
      _wallet.LoadOrdersFromBroker();
```

```
   }

   delete(currentOpenOrders);
}

void UpdateOrders()
{
   // Print("Nr of open orders for update: " + _wallet.GetOpenOrders().Count());

   _wallet.GetOpenOrders().Rewind();
   while(_wallet.GetOpenOrders().HasNext())
   {
      Order* order = _wallet.GetOpenOrders().Next();
      double pipsProfit = order.CalculateProfitPips();
      order.CurrentProfitPips = pipsProfit;

      if (pipsProfit<order.LowestProfitPips)
      {
         order.LowestProfitPips = pipsProfit;
      }
      else if (pipsProfit > order.HighestProfitPips)
      {
         order.HighestProfitPips = pipsProfit;
      }
   }
}

bool ExecutePendingCloseOrders()
{
   OrderCollection* pendingCloseOrders = _wallet.GetPendingCloseOrders();
   if (pendingCloseOrders.Count() == 0)
   {
     return true;
   }
```

```
   bool totalSuccess = true;

   bool orderSuccess = true;
   for (int i = pendingCloseOrders.Count()-1; i >= 0; i--)
   {
     if (!orderSuccess)
     {
       totalSuccess = false;
     }

     orderSuccess = false;
     Order* pendingCloseOrder = pendingCloseOrders.Get(i);

     bool success = OrderRepository::CloseOrder(pendingCloseOrder);
     if (success)
     {
       Order* closedOrder = OrderRepository::GetLastClosedOrder();
       pendingCloseOrder.CloseTime = closedOrder.CloseTime;
       delete (closedOrder);

       _wallet.MovePendingCloseToClosedOrders(pendingCloseOrder);

       orderSuccess = true;
     }
     else
     {
       Print("CloseOrder failed!");
     }
   }

   return totalSuccess;
 }

bool ExecutePendingOpenOrders()
{
```

```
if (_wallet.GetPendingOpenOrders().Count() == 0)
{
   return true;
}

bool successTotal = true;

// Open pending open orders
OrderCollection* pendingOpenOrders = _wallet.GetPendingOpenOrders();
bool successOrder = true;
for (int i = pendingOpenOrders.Count() - 1; i >= 0; i--)
{
   if (!successOrder)
   {
      successTotal = false;
   }

   successOrder = false;
   bool isTradeContextFree = false;
   double StartWaitingTime = GetTickCount();

   while (true)
   {
      if (IsTradeAllowed())
      {
         isTradeContextFree = true;

         // refresh the market information
         RefreshRates();
         break;
      }

      int MaxWaiting_sec = 10;
      // if the expert was terminated by the user, stop operation
      if (IsStopped())
```

```
   {
      Print("The expert was stopped by the user!");
      break;
   }

   // if it is waited longer than it is specified in the variable named
   // MaxWaiting_sec, stop operation, as well
   if (GetTickCount() - StartWaitingTime > MaxWaiting_sec * 1000)
   {
      Print("The standby limit(" + DoubleToStr(MaxWaiting_sec) + " sec) exceeded!");
      break;
   }

   Sleep(100);
}

Order* order = pendingOpenOrders.Get(i);
if (!isTradeContextFree)
{
   _wallet.CancelPendingOpenOrder(order);
   continue;
}

bool success = OrderRepository::OpenOrder(order);
if (success)
{
   // Move order to OpenOrders
   _wallet.MovePendingOpenToOpenOrders(order);

   // reset comments
   stopLossComment = "";
   takeProfitComment = "";

   successOrder = true;
}
```

```
            else
            {
               // Wallet remove pending open order, on next tick it will be evaluated if signal is still valid
               _wallet.CancelPendingOpenOrder(order);
               successOrder = false;
            }
         }

         return successTotal;
      }
};

//+------------------------------------------------------------------+
//|  Expert Initialization Function                                  |
//+------------------------------------------------------------------+

int OnInit()
{

            SetPipPoint();
            if (PipPoint == 0)
            {
                        Print("Couldn't find correct point for symbol.");

               return (INIT_FAILED);
            }

            _ea = new EA();
            _ea.Init();



            return (INIT_SUCCEEDED);

}
```

```
//+------------------------------------------------------------------+
//|  Expert Deinitialization Function                                |
//+------------------------------------------------------------------+

void OnDeinit(const int reason)
{
        delete(_ea);
}

//+------------------------------------------------------------------+
//|  Expert Advisor Function                                         |
//+------------------------------------------------------------------+

void OnTick()
{
        _ea.HandleTick();
}
```