OpenQuant API References

10.10.2016
SmartQuant Ltd
Anton Fokin

Framework Classes

Framework

The Framework class is the core class of the SmartQuant Framework. It provides access to framework managers, such as DataManager, OrderManager, PortfolioManager, etc.

When an object of the Framework class is created, it reads framework Configuration file and initializes framework managers according to Configuration settings. It also loads object streamers, execution and data providers listed in the Configuration file.

Usually you create Framework with

Framework framework = new Framework("MyFramework");

constructor, where "MyFramework" is the name of the framework.

You can use Framework.Current static member to access the current framework instance. If there is no framework created, a new framework will be created and Framework.Current will point to this newly created framework. Note that since Framework.Current is a static member, you should use this member with care in applications where you may have multiple frameworks.

You should always call Framework.Dispose() method in the end of your program to properly close database files and release other resources, otherwise you can lose imported data or part of your strategy output, or incorrectly disconnect from a broker.

Configuration

The Configuration class stores configuration settings loaded from the configuration.xml file. The configuration settings can be accessed using Framework.Configuration property.

Installation

The Installation class provides information about default installation directories of applications built with the SmartQuant Framework, such as Installation.DataDir, Installation.ConfigDir and so on.

Clock Classes

Clock

This class represents Framework Clock. The main idea behind Clock class is that time definition is different in the simulation and realtime modes. In the simulation mode the current time is equal to the local system time, while in the simulation mode the current time is defined by the time stamp of the latest simulated market data. The Clock.DateTime property of the Clock class provides such a reference to the current framework time that can be used identically in both modes.

That's it. Whenever you need to get the current time in your trading strategy or application code, simply use Framework.Clock.DateTime and it will work in both simulation and real time modes without any changes in the code.

You rarely need to create an object of the Clock class yourself since framework clock object is created and initialized in the Framework constructor and you can always access it using Framework.Clock property. Framework manager classes and strategy classes provide quick access to Framework.Clock via their corresponding Clock property, so that for example you can simply use Clock.DateTime when you override strategy methods.

The Clock class is also used to manage framework reminders. A reminder can be added and removed using corresponding Clock.AddReminder and Clock.RemoveReminder methods.

Reminder

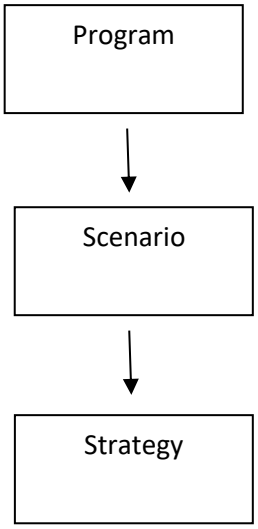
A reminder is an object with associated remainder callback method, which triggers on certain datetime set in the reminder constructor.

Strategy classes

Scenario

As we already know, OpenQuant 2014 algo trading strategy is a normal Microsoft Visual Studio solution that can be compiled into a standalone executable application.

The common structure of SmartQuant Strategy Solution is shows on the figure below.



Program.cs creates application execution environment and runs strategy Scenario, calling Scenario.Run() method.

```
using System;

using SmartQuant;

namespace OpenQuant
{
    class Program
    {
        static void Main(string[] args)
        {
            Scenario scenario = new MyScenario(Framework.Current);

            scenario.Run();
        }
    }
}
```

In turn, Scenario.Run() method (overridden by user in MyScenario class inherited from Scenario class) creates a strategy, sets its properties, defines execution environment and runs the strategy in backtest, paper or live mode. There can be other, much more complex scenarios, such as Optimization scenarios, Monte Carlo or Walk-Forward scenarios. The scenario below backtests MyStrategy with AAPL instrument on 60 second time bars, and tells the market DataSimulator to simulate market data from 2012/12/16 till 2012/12/20.

```
using System;

using SmartQuant;

namespace OpenQuant
{
    public class MyScenario : Scenario
    {
        public MyScenario(Framework framework)
            : base(framework)
        {
        }

        public override void Run()
        {
            strategy = new MyStrategy(framework, "Backtest");

            Instrument instrument = InstrumentManager.Instruments["AAPL"];

            strategy.AddInstrument(instrument);

            DataSimulator.DateTime1 = new DateTime(2012, 12, 16);
            DataSimulator.DateTime2 = new DateTime(2013, 12, 20);

            BarFactory.Add(instrument, BarType.Time, 60);

            StartStrategy(StrategyMode.Backtest);
        }
    }
}
```

Strategy

Historically, SmartQuant was the first trading software company that offered event driven approach to algo strategy development back in 1997. Before this strategy developers used to program strategy logic in a “for” loop similar to

```
for(int i=0;i<bars.Count;i++)
{
  ….

  If (bars[i].Close > bars[i-1].Close)
    Buy(“MSFT”)
}
```

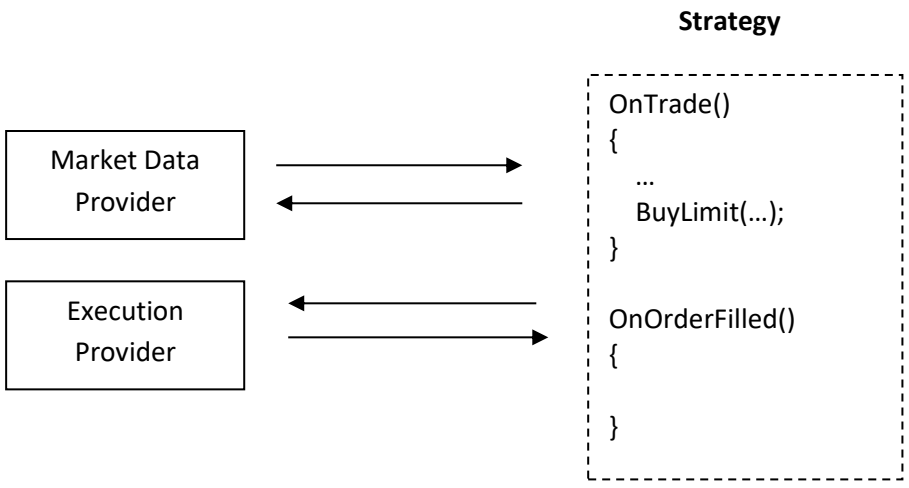
Although this idea worked relatively well in the early versions of trading software such as TradeStation or WealthLab, there are several problems with this approach. First of all, strategy developers have full access to future data (notice that we access array of bars that contains the entire collection of historical bars used in simulations). Most importantly, it’s not easy to switch from strategy simulations to live strategy trading without changing strategy code. There is simply no collection of bars that you can loop over during live trading. Instead, a strategy receives a flow of live data and execution events, which is quite different from static collection of data that can be enumerated in the “for” loop.

In order to reflect actual process of live trading and decision making, SmartQuant has developed event driven approach to algo strategy programming. Actually the idea is quite simple. Programmatically speaking, a strategy is a class that has several methods corresponding to events occurring during real world trading. According to event driven programming terminology these methods are called event handlers and their names usually look like OnXXX (such as OnBid) indicating that these method are called upon specific events (naturally OnBid method is called when a new bid is published by exchange). These methods are marked as virtual and can be overridden by a strategy developer to react on specific events.

```
public override void OnBar(Bar bar)
{
   bars.Add(bar);

   if (bars[i].Close > bars[i-1].Close)
      Buy("MSFT")
}
```

Our strategy logic code doesn't see a new bar until it actually comes into OnBar event handler, so that we cannot mistakenly use future data during decision making. This code works identically with live bars coming from market data provider such as IQFeed, or with simulated historical data. Thus we can switch from strategy backtesting to paper or live trading in production with one mouse click.



InstrumentStrategy

SellSideStrategy

SellSideInstrumentStrategy

Stop Classes

StopType

Enumerator that defines the type of a stop object

Fixed
Trailing
Time

StopMode

Enumerator that defines the mode of a stop object

Absolute
Percent

StopFillMode

Enumerator that defines how the stop object should be filled if strategy uses bar data

Close
HighLow
Stop

StopStatus

Enumerator that indicates the current status of a stop object

Active
Executed
Canceled

Stop

A stop object attached to a position in a strategy.

Data Classes

Tick

This class represents a generic tick market data (price and size). Tick is the base class for specific tick data classes (Bid, Ask, Trade)

TickType

Enumerator that defines type of tick

Bid
Ask
Trade

Bid

The bid price represents the maximum price that a buyer or buyers are willing to pay for a security.

Ask

The ask price represents the minimum price that a seller or sellers are willing to receive for the security.

Trade

A trade occurs when the buyer and seller agree on a price for the security.

Trade class also has Trade.Direction property that indicates direction of this trade, which can be given by market data provider or calculated.

Direction

Constant members of this class indicate direction of a trade

Undefined = -1;
Plus = 0;

ZeroPlus = 1;
Minus = 2;
ZeroMinus = 3;

Quote

The Quote (or a two-way quote) is a type of quote that gives both the bid and the ask price of a security, informing traders of the current price at which they could buy or sell the security. The two-way quote also shows the spread between the bid and the ask, giving traders an idea of the current liquidity in the security (a smaller spread indicates more liquidity).

Bar

A bar is a representation of a price movement that contains the open, high, low and closing prices for a set period of time or a specified set of data.

A bar also holds Bar.Fields ObjectTable that can be used to store any type of data entries that have corresponding streamers registered in the framework

Level2

FundamentalData

Constant members of this class indicate several standard types of fundamental data. You can use this helper class to write Fundamental[FundamentalData.Beta] = 1.34;

CashFlow = 1;
PE = 2;
Beta = 3;
ProfitMargin = 4;
ReturnOnEquity = 5;
PriceBook = 6;
DebtEquity = 7;
InterestCoverage = 8;
BookValue = 9;
PriceSales = 10;
DividendPayout = 11;
Split = 12;

Fundamental

This class holds Fundamental.Fields ObjectTable that can be used to store any type of fundamental data entries that have corresponding streamers registered in the framework, for example Fundamental[3] = 1.34 (which is the same as Fundamental[FundamentalData.Beta] = 1.34).

News

This is one news record for an Instrument.

The news record usually has urgency, url, headline and text.

OrderBook

OrderBookAggr

TickSeries

This is a collection of tick objects ordered by their time stamps.

TimeSeriesItem

This class represents one entry of TimeSeries and holds DateTime / double Value pair. TimeSeriesItem is derived from DataObject and has a corresponding streamer in the framework.

TimeSeries

This is a time series of double values. Internally this is a collection of TimeSeriesItems ordered by their DateTime property.

TimeSeries.Add method adds TimSeriesItem to this series. Since TimSeries is an ordered collection of TimSeriesItems, this item will either be added to the end of time series or inserted in correct place in the collection depending on its DateTime and DateTime of the last item in the series. Since insert operation is much more time consuming than add operation, keep this in mind when you add a large number of items and want to achieve good performance.

TimeSeries has a list of associated Indicators. These indicators get updated once an item is added to the TimeSeries.

BarSeries

This is a series of Bars.

Similar to TimseSeries, BarSeries has a list of associated Indicators. These indicators get updated once a bar is added to the TimeSeries.

BarFactory

BarFactory builds bars from ticks (bids, ask or trades) on the fly and emits these bars into the framework, and, consequently, into your strategy.

BartFactory holds a list of BarFactoryItems.

BarFactoryItem

A BarFactoryItem builds bars for specific Instrument, BarType (Time, Tick, Volume, Range, Session), BarInput (Trade, Bid, Ask, Middle, Tick, BidAsk) and Size.

Users can override OnData method of BarFactoryItem class in derived classes to implement custom bar types.

BarType

Enumerator that defines type of bar

Time
Tick
Volume
Range
Session

BarInput

Enumerator that defines what types of tick data should be used to build bars in the BarFactory

Trade
Bid
Ask
Middle
Tick

BidAsk

BarData

Enumerator that defines data field (or calculated combination of data fields) of a bar

Close
Open
High
Low
Median
Typical
Weighted
Average
Volume
OpenInt
Range
Mean
Variance
StdDev

BarStatus

Enumerator that indicates the current status of a bar

Incomplete,
Complete,
Open,
High,
Low,
Close

BarFilterItem

This class holds bar type – bar size pair. BarFilter uses a list of BarFilterItems to filter bars with particular types and sizes.

BarFilter

Filters bars with particular bar types and sizes.

BarFilter is used in the ExecutionSimulator to define which bar types and sizes should be used by the execution simulator to fill market orders and trigger limit and stop orders. For example if you have one minute, ten minute and hourly bars among historical data that you use in simulations, then most likely you want to simulate order execution with the shortest (one minute) bars and skip all other (ten minute and hourly) bars. You can do this with ExecutionSimulator.BarFilter.Add(BarType.Time, 60).

DataFilter

DataSeries

A series of DataObjects that can be stored in a DataFile.

DataFile

This is an object oriented database.

Objects of classes derived from DataObject class, DataSeries of DataObjects and any objects, which have corresponding Streamers can be stored in the DataFile.

NetDataSeries

This is a remote version of DataSeries that can be stored in remote DataFile on the DataFileServer and accessed over TCP/IP network.

NetDataFile

This is a remote version of DataFile that can access DataFile on the DataFileServer over TCP/IP network.

DataManager

DataServer

Indicator Classes

Indicator

This is the base class for technical indicators.

Indicator class constructor takes input ISeries (TimeSeries, TickSeries, BarSeries) as parameter and adds this indicator to the list of indicators attached to input ISeries. Once a new entry is added to this input series, Indicator.Calculate() method is called for all attached indicators if they have Indicator.AutoUpdate property set to true. These way indicators get automatically updated when a new entry is added to their input series.

Users can override Indicator.Calculate() method in derived classes to develop their own technical indicators.

SMA, BBU, RSI, ...

The SmartQuant Framework offers a large number of various technical indicators, which can be found in the SmartQuant.Indicators namespace.

Instrument Classes

AltId

Different market data providers and brokers (execution providers) can define financial instruments differently in their systems. For example AAPL stock corresponds to AAPL US symbol in Bloomberg system and AAPL.OQ symbol in Reuters system. In order to store and provide correct symbol and other information (exchange, currency) to particular provider, we introduce AltId class, which provides "Alternative Identification" information.

AltIdList

This is a collection of AltId objects indexed by ProviderId for convenience, see AltIdList.GetByProviderId();

Instrument

This class represents a financial instrument and holds its definitions and properties, such as Type, Symbol, Currency, Exchange, TickSize, Strike, Expiration and so on, as well as a list of alternative definitions for this instrument Instrument.AltId.

It also contains a list of Instrument Legs, Instrument.Legs, in case of multi-leg instrument.

It provides quick access to the most recent pricing information for this instrument, Instrument.Trade, Instrument.Bid, Instrument.Ask, Instrument.Bar.

It has a reference to ObjectTable, Instrument.Fields, where any additional information related to this instrument can be stored by a user.

InstrumentList

This is a collection of instruments indexed by instrument.Symbol and Instrument.Id for convenience, see InstrumentList.GetBySymbol(), InstrumentList.GetById(). You can also use InstrumentList[“AAPL”] to get instrument by its symbol.

InstrumentManager

This class manages a list of instruments created in the framework.

It also manages instruments stored in instrument data base using InstrumentServer.

InstrumentServer

This is the base class that manages instrument storage. It provides methods to get, save and delete instruments in the instrument data base. InstrumentServer is used by the InstrumentManager to load InstrumentList of all available instruments from instrument data base on framework startup.

FileInstrumentServer

This class inherits from InstrumentServer class to use local or remote DataFile as instrument data base.

Execution Classes

ExecutionMessage

This is a message sent to or received from a broker API or FIX via ExcutionProvider.

ExecInst

Constant members of this class define execution instruction of an execution command. These definitions follow FIX standard.

NotHeld = '1';
Work = '2';
GoAlong = '3';
OverTheDay = '4';
Held = '5';
ParticipateDontInitiate = '6';
StrictScale = '7';
TryToScale = '8';
StayOnBidside = '9';
StayOnOfferside = '0';
NoCross = 'A';
OkToCross = 'B';
CallFirst = 'C';
PercentOfVolume = 'D';
DNI = 'E';
DNR = 'F';
AON = 'G';
InstitutionsOnly = 'I';
LastPeg = 'L';
MidPricePeg = 'M';
NonNegotiable = 'N';
OpeningPeg = 'O';
MarketPeg = 'P';
PrimaryPeg = 'R';
Suspend = 'S';
FixedPeg = 'T';
CustomerDisplayInstruction = 'U';
Netting = 'V';
PegToVWAP = 'W';

ExecutionCommandType

Enumerator that defines the type of an execution command

Send
Cancel
Replace

ExecutionCommand

This is an execution message with instructions (command) sent to a broker (place a new order, request to cancel or replace the order).

ExecType

Enumerator that indicates the type of an execution report. These definitions follow FIX standard.

ExecNew
ExecStopped
ExecRejected
ExecExpired
ExecTrade
ExecPendingCancel
ExecCancelled
ExecCancelReject
ExecPendingReplace
ExecReplace
ExecReplaceReject
ExecTradeCorrect
ExecTradeCancel
ExecOrderStatus
ExecPendingNew
ExecClearingHold

ExecutionReport

This is an execution message received from a broker in response to an execution command (order acceptance, rejection, fill).

OrderSide

Enumerator that indicates the side of an order

Buy
Sell

OrderType

Enumerator that indicates the type of an order

Market
Stop
Limit
StopLimit
MarketOnClose
Pegged
TrailingStop
TrailingStopLimit

OrderStatus

Enumerator that indicates the status of an order.

NotSent
PendingNew
New
Rejected
PartiallyFilled
Filled
PendingCancel
Cancelled
Expired
PendingReplace
Replaced

TimeInForce

Enumerator that indicates time in force of an order

ATC
Day
GTC
IOC
OPG
OC
FOK
GTX
GTD
GFS
AUC

Order

This is a class that holds information about the current status and life circle of an order sent to a broker.

OrderManager

OrderServer

Currency Classes

CurrencyId

Holds currency identifiers, for example CurrencyId.USD, and performs conversion between currency names and identifiers, CurrencyID.GetId(string name), CurrencyId.GetName(byte id).

CurrencyConverter

Converts amount of currency to another currency with CurrencyConverter. Convert(double amount, byte fromCurrencyId, byte toCurrencyId) method.

Since there is no generic mechanism suggesting how currency conversion should be performed and where currency exchange rates should be acquired in every specific case, this base class actually doesn't perform any conversion and simply returns the same amount of currency. A user should override its Convert method to implement specific currency conversion mechanism. Such user defined currency converter can be set as the default currency converter of the framework with Framework.CurrencyConverter property. Once set, this default currency converter will be used everywhere in the framework where currency conversion is required, for example in the Portfolio.Account to convert values of account positions to the base currency of this account.

Portfolio Classes

Fill

A Fill is the action of fully or partially completing or satisfying an Order to buy or sell a financial Instrument. Technically speaking a Fill is created and added to Order.Portfolio when an ExecutionReport with ExecType.Trade is received from a broker for this Order.

Transaction

A Transaction occurs when an order becomes completely filled by the broker. Consequently the Transaction contains one or more Fills, Transaction.Fills.

PositionSide

Enumerator that indicates the side of a portfolio Position

Long
Short

Position

A Position is the amount of financial Instrument hold by a trader or trading Strategy in Portfolio.

Position has a quantity, Position.Qty, which is always positive, and position side, Position.Side, which can be long or short. Position.Amount is positive for long positions and negative for short positions, which means that Position.Amount is equal to Position.Qty for long positions and equal to –Position.Qty for short positions.

AccountTransaction

AccountTransaction is created and added to the list of Account.Transactions when amount of currency is deposited to or withdrawn from Account.

AccountPosition

This class represents a position in specific currency in the Account.

Account

Account holds a list of account positions, Account.Positions, and account transactions, Account.Transactions. When amount of currency is added to or withdrawn from Account, an AccountTransaction is created and added to the list of account transactions and a corresponding currency AccountPosition is created or updated accordingly.

Account has its base currency, identified by Account.CurrencyId. Account value, Account.Value is the sum of values of account positions converted to account base currency using Framework.CurrencyConverter

Portfolio

A Portfolio can be seen as a group of Positions hold by a trader or trading Strategy. Portfolio Positions are built up from ExecutionReports or Fills added to the Portfolio.

Portfolio has an associated Account, Portfolio.Account, which is used for cash balancing when a new Fill gets added to the Portfolio.

Portfolio has Portfolio.Parent property that can point to a parent portfolio of this portfolio. Portfolio.Children property holds a list of children of the parent portfolio. A parent portfolio can be a child of another portfolio. This way a group of portfolios can form a portfolio tree. Changes in child portfolio are propagated to its parent portfolio, which means that if a fill is added to the child portfolio, this fill is also added to its parent portfolio. This way positions in the parent portfolio can be seen as consolidated positions and the parent portfolio can be seen as consolidated portfolio.

PortfolioManager

This class manages a list of portfolios created in the framework.

It also manages portfolios stored in portfolio data base using ProtfolioServer.

PortfolioServer

This is the base class that manages portfolio storage. It provides methods to get, save and delete portfolios in the portfolio data base. PortfolioServer is used by the PortfolioManager to load PortfolioList of all available portfolios from portfolio data base on framework startup.

FilePortfolioServer

This class inherits from PortfolioServer class to use local or remote DataFile as portfolio data base.

Output Classes

Group

GroupEvent

GroupManager

Provider Classes

Provider

IDataProvider

IExecutionProvider

IHistoricalProvider

INewsProvider

IFundamentalProvider

IBTWS, PATS

ProviderError

This event holds information sent by a Provider in case of error, warning or notification.

A Strategy has virtual OnProviderError(ProviderError error) method which can be overridden to react in response to provider error event.

ProviderList

ProviderManager

Simulation Classes

DataProcessor

DataSimulator

RealTimeDataSimulator

CommissionType

Enumerator that indicates the type of commission

PerShare
Percent
Absolute

CommissionProvider

SlippageProvider

ExecutionSimulator

Use ExecutionSimulator. BarFilter to define which bar types and sizes should be used by the execution simulator to fill market orders and trigger limit and stop orders For example if you have one minute, ten minute and hourly bars among historical data that you use in simulations, then most likely you want to simulate order execution with the shortest (one minute) bars and skip all other (ten minute and hourly) bars. You can do this with ExecutionSimulator.BarFilter.Add(BarType.Time, 60).

Serialization Classes

Streamer

StreamerManager

Optimization Classes

OptimizationParameter

OptimizationParameterSet

OptimizationUniverse

Optimizer

MulticoreOptimizer

SimulatedAnnealingOptimizer

GeneticOptimizer

OptimizationManager