Developing Algo Trading Strategies with SmartQuant Framework
The Getting Started Guide

30.12.2014
SmartQuant Ltd
Dr. Anton B. Fokin
Introduction

You can develop stand alone financial applications with SmartQuant Framework. Such applications can range from market data analysis applications to trading front ends and order execution servers. You can also develop automated trading strategies as standalone applications right in MS Visual Studio. This was not possible before.

Prerequisites

Application development with SmartQuant Framework requires Microsoft .NET 4.5.1 and Microsoft Visual Studio 2012 or 2013. If you have Visual Studio 2012, most likely you need to install .NET Framework 4.5.1 Developer Pack from http://www.visualstudio.com/downloads/download-visual-studio-vs

Installing SmartQuant Framework

The SmartQuant Framework is usually installed together with OpenQuant application.

OpenQuant applications installs in the standard Program Files\SmartQuant Ltd\OpenQuant 2014 folder. The data files (data.quant and instruments.quant) can be found in your AppData\Roaming\SmartQuant Ltd\OpenQuant 2014\data folder, configuration files are in the corresponding folders in the AppData\Roaming\SmartQuant Ltd\OpenQuant 2014, and the demo solutions can be found in your Documents\OpenQuant 2014\Solutions folder.
Reinstalling SmartQuant Framework

It’s always a good idea to perform “clean” uninstall before installing a new version. OpenQuant installer doesn’t override user files such as data files, configuration files or strategy files. Thus you may miss important changes in those files if you don’t remove them manually.

In order to perform “clean” uninstall, you should uninstall OpenQuant selecting Uninstall in windows start menu,

and then manually remove files from your AppData\Roaming\SmartQuant Ltd\OpenQuant 2014\ and your Documents\OpenQuant 2014\Solutions\ folders . Indeed you can leave or backup files that you need. For example you don’t need to delete your own strategies if you need them, but if you want to update demo strategies, you need to delete corresponding files.

Then you can install a new version of OpenQuant 2014 as discussed in the previous chapter.

Event Driven Strategy Development

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

```csharp
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 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.

```csharp
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.

The best way to understand a new concept is to learn by example.

Creating Strategy Project in OpenQuant 2014
We assume that you have successfully installed OpenQuant 2014 on your computer.

Navigate to File->New->Solution... in main OpenQuant menu.

The New Solution dialog should pop up. Select SmartQuant Instrument Strategy Solution and name it MySolution.

Navigate to Solution Explorer window in OpenQuant IDE. If you can’t find the Solution Explorer window, navigate to View in the main OpenQuant menu and select Solution Explorer.
Strategy Solution Structure

We have mentioned many times in this and other OpenQuant 2014 documents that OpenQuant 2014 algo trading strategy is a normal Microsoft Visual Studio solution that can be compiled into a standalone executable application.

Let’s review the structure of SmartQuant Strategy Solution.
MyStrategy solution consists of two projects: Backtest and MyStrategy. Backtest is a project that contains Program.cs file. You can mark this project as StartUp project and then it will be executed when you click Run button in OpenQuant main menu.

Program.cs creates application execution environment and runs a strategy scenario.

```csharp
using System;
using SmartQuant;

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

Usually a scenario 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 or walk-forward scenarios. You will learn about scenarios in the corresponding chapter of this doc. For now it’s enough to say that MyScenario 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. This scenario should work with demo instruments and data supplied with OpenQuant 2014 installation, so that it should compile and run out of the box.

MyScenario.cs
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 defines strategy execution logic and this is the class we are currently interested in.

MyStrategy.cs

using System;
using SmartQuant;

namespace OpenQuant
{
    public class MyStrategy : InstrumentStrategy
    {
        public MyStrategy(Framework framework, string name)
            : base(framework, name)
        {
        }

        protected override void OnStrategyStart()
        {
        }

        protected override void OnBar(Instrument instrument, Bar bar)
        {
        }
    }
}
The default strategy template includes two OnXXX event handlers - OnStrategyStart and OnBar. Let’s run an experiment and add Console.WriteLine to these methods to see when these event handlers get called.

**Understanding DataSimulator**

The DataSimulator should behave as a realistic data provider.

- Your strategy should operate in the same way during backtest and live trading. 99% of market data providers do not offer bar data in their data feeds but rather tick and (optionally) market depth data.

- Simulations of your strategy order fills should be as realistic as possible. It’s hard to simulate realistic fills with bar data. Even tick by tick trade data can produce pretty unrealistic and misleading results. You should use bid/ask data feed if you want to get a good feeling of how your strategy behaves during live trading.

- While it’s ok to calculate your strategy entry signals using bar data, most likely you want to use tick data to trigger take profit and stop loss exits.

**Understanding BarFactory**

Usually market data providers do not supply bar data in their market data feeds. Even if your data provider offers a possibility to request bar data, most likely your strategy would need non-standard bars or bars created according to your own custom rules.

The BarFactory is used to create bars (candlesticks) from incoming tick stream on the fly. Then these bars get emitted into the framework and you can process them in OnBarOpen and OnBar event handlers in your strategy.

BarFactory can create time, tick and volume bars out of the box. It also provides a mechanism to create user defined bar types by inheriting BarFactoryItem class.

**Data Simulation Example**

OpenQuant 2014 setup comes with a small sample of market data that we will use in our demo. Let’s explore this data set first. Navigate to the Instrument window in OQ 2014 IDE and double click on MSFT instrument. You should see Data Explorer window for MSFT instrument with bid, ask and trade series. If you click on one of these series, you should see it content in the tab below:
We will use MSFT instrument in our data simulation example.

As the first exercise we create an empty OQ2014 solution with InstrumentStrategy project.

We add OnBid, OnAsk and OnTrade event handlers to MyStrategy code and print all incoming data to see what we get when we run our experiment.
using System;
using SmartQuant;

namespace OpenQuant
{
    public class MyStrategy : InstrumentStrategy
    {
        public MyStrategy(Framework framework, string name)
        :
            base(framework, name)
        {
        }

        protected override void OnStrategyStart()
        {
        }

        protected override void OnBid(Instrument instrument, Bid bid)
        {
            Console.WriteLine(bid);
        }

        protected override void OnAsk(Instrument instrument, Ask ask)
        {
            Console.WriteLine(ask);
        }

        protected override void OnTrade(Instrument instrument, Trade trade)
        {
            Console.WriteLine(trade);
        }

        protected override void OnBar(Instrument instrument, Bar bar)
        {
            Console.WriteLine(bar);
        }
    }
}

Now we add MSFT instrument to our strategy in the scenario file

using System;
using SmartQuant;

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

        public override void Run()
        {
            strategy = new MyStrategy(framework, "Backtest");
            Initialize();
            strategy.AddInstrument("MSFT");
            StartStrategy(StrategyMode.Backtest);
        }
    }
}
and start our strategy in the backtest mode. We run our example in Microsoft Visual Studio as a console application, so that we get strategy output in the system console window. In OQ2014 IDE you should see the same in the Output window.

As you can see our strategy prints bids, asks and trades – all available historical market data that we have in the demo data sample. That’s it. Our experiment shows that by default the DataSimulator simulates all standard data series available for an instrument in the local OQ2014 database.

Sometimes we don’t want to simulate our strategy with all market data that we downloaded into the database. Usually we want to use a specific simulation interval, i.e. run strategy backtest from DateTime1 to DateTime2. In order to do so in our example, we should set simulation interval using DataSimulator DateTime1 and DateTime2 properties.

We add the following lines into our scenario file

```csharp
DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
DataSimulator.DateTime2 = new DateTime(2013, 12, 19);
```

and run the backtest
We see that simulated market data stream now starts with 18th of December 2013.

The next step is to filter (turn on and off) data series that we want to use in simulations. Assume that we don’t need precise simulation of order execution and we want to speed up backtest and replay trade data only, skipping bids and asks for now. We can do this using DataSimulator.SubscribeXXX property, where XXX stands for standard market data types, such as Bid, Ask, Trade, Bar, Level2, etc. In our case the following two lines

```
DataSimulator.SubscribeBid = false;
DataSimulator.SubscribeAsk = false;
```

should do what we want.
Now we don’t have bids and asks in our simulated market data feed any more.

Turning single data type simulation on and off is easy, but dealing with bars is a bit more troublesome since we may easily have several bar series with bars of different types and sizes for a single instrument stored in the market data base. Moreover, we can replay series of bars directly from the database, or we can build bars on the fly from streams of trades, bids or asks that we replay from the database.

The latter we can do using the BarFactory and now we will try to simulate one minute bars that we create from trade feed that we have just managed to get in our last experiment.

The following line should do the job

```csharp
BarFactory.Add("MSFT", BarType.Time, 60);
```

And complete scenario file now looks like

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework) : base(framework)
        {
        }

        public override void Run()
        {
            strategy = new MyStrategy(framework, "Backtest");
            Initialize();
            DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
            DataSimulator.DateTime2 = new DateTime(2013, 12, 19);
            DataSimulator.SubscribeBid = false;
            DataSimulator.SubscribeAsk = false;
            BarFactory.Add("MSFT", BarType.Time, 60);
            strategy.AddInstrument("MSFT");
            StartStrategy(StrategyMode.Backtest);
        }
    }
}
```
It's a rather bad programming style to write “MSFT” symbol as method parameter in several places in the code, so let's improve it a bit and add

```csharp
Instrument instrument = InstrumentManager.Instruments["MSFT"];
```

line.

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework)
            : base(framework)
        {
        }

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

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

            DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
            DataSimulator.DateTime2 = new DateTime(2013, 12, 19);
            DataSimulator.SubscribeBid = false;
            DataSimulator.SubscribeAsk = false;

            BarFactory.Add("MSFT", BarType.Time, 60);
            strategy.AddInstrument(instrument);
            StartStrategy(StrategyMode.Backtest);
        }
    }
}
```
We can add several items into the BarFactory for the same instrument, for example we can simulate 1min and 5min bars. In order to achieve this we just need to add one more line

```csharp
BarFactory.Add(instrument, BarType.Time, 300);
```

Let’s comment out `Console.WriteLine` in the OnTrade event handler in the strategy code, so that our strategy prints bars and nothing else. If we now run backtest again, the output should look like

```
You see five one minute bars followed by one five minute bar, which seems to be correct.
```

Out of the box the BarFactory knows how to create Time, Tick and Volume bars. You can change BarType using the second parameter of the BarFactory.Add method. For example a request to create bars with 100 ticks each is

```csharp
BarFactory.Add(instrument, BarType.Tick, 100);
```

There is also an optional parameter in the BarFactory.Add method that indicates what type of incoming ticks the BarFactory should use to create bars. Usually we create bars from trades, but from time to time it becomes important to be able to create bars from asks or bids. For example FX feeds do not provide trade information. In this case you can write

```csharp
BarFactory.Add(instrument, BarType.Time, 60, BarInput.Ask);
```

to create bars from asks. You can also indicate BarInput.Bid or BarInput.Middle. The latter stands for the middle price between best bid and ask. Indeed the DataSimulator should be subscribed for bids or/and asks and you should have bid or/and ask data in OQ2014 database to create bars from bids and asks. The default value of this parameter is BarInput.Trade.

Now we know how to create bars from tick data on the fly. Although usually this is the right way to get bars into your strategy, you may also want to speed up simulations and replay bars stored in the local database instead of replaying huge amount of tick data. Let’s see how we deal with this in OQ2014.
First of all we have to store bar data in OQ database. Indeed we can import bars from a text file or compress bars from ticks using OQ Compress dialog in the DataExplorer window, but here we will show how to use an alternative way to prepare bar series for further simulations. We will use our strategy and save bars created by the BarFactory in OnBar event handler.

```csharp
protected override void OnBar(Instrument instrument, Bar bar)
{
    DataManager.Save(bar);
}
```

Note that you should check that your program properly disposes the framework when your application exits. This is a must if you want to store data in the local database. Framework.Dispose flushes data on disk and closes database files. Most likely your data will not be saved or will become corrupted if you don’t call Framework.Dispose in the end of your program.

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    class Program
    {
        static void Main(string[] args)
        {
            Scenario scenario = new MyScenario(Framework.Current);
            scenario.Run();
            Framework.Current.Dispose();
        }
    }
}
```

Let's first record one minute created with the BarFactory. In order to do so we comment out the code line that creates five minute bars in the scenario and run simulations

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework)
            : base(framework)
        {
        }

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

            Instrument instrument = InstrumentManager.Instruments["MSFT"];
            DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
        }
    }
}
```
DataSimulator.DateTime2 = new DateTime(2013, 12, 19);
DataSimulator.SubscribeBid = false;
DataSimulator.SubscribeAsk = false;

BarFactory.Add(instrument, BarType.Time, 60);
//BarFactory.Add(instrument, BarType.Time, 300);
strategy.AddInstrument(instrument);
StartStrategy(StrategyMode.Backtest);
}
}

Then we run OpenQuant IDE and check what data series we have now for MSFT instrument

![Image of OpenQuant IDE](image)

We should see a new Bar.Time.60 series.

Now let’s comment DataManager.Save() line, otherwise we will be adding the same bars over and over again with every simulation run.

We run simulations and look at the output window
It's easy to see that we receive the same bar twice now. One bar comes from the BarFactory and the other one comes directly from the bar series that we have recorded in the previous exercise.

Now we can tell DataSimulator to skip trades as well as bids and asks and comment out BarFactory.Add.

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework) : base(framework)
        {
        }

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

            Initialize();

            Instrument instrument = InstrumentManager.Instruments["MSFT"];
            DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
            DataSimulator.DateTime2 = new DateTime(2013, 12, 19);

            DataSimulator.SubscribeBid = false;
            DataSimulator.SubscribeAsk = false;
            DataSimulator.SubscribeTrade = false;

            //BarFactory.Add(instrument, BarType.Time, 60);
            //BarFactory.Add(instrument, BarType.Time, 300);

            strategy.AddInstrument(instrument);
            StartStrategy(StrategyMode.Backtest);
        }
    }
```
And run simulations again. This time we only replay bars from the database. Simulations end much quicker now and we don’t see repeating bars in the program output anymore.

So now we know how to simulate a single bar series stored in the local OQ database. Let’s move on and create another bar series for the same instrument.

We set DataManager.SubscribeTrade = true to playback trades, we uncomment the line that requests BarFactory to create 5min bars and comment a line that creates 1min bars. We also uncomment DataManager.Save() in the OnBar.

Now we have a bar series in the local database and this means that the data simulator will playback these bars and we will save them again in the OnBar. This is not what we want indeed, so that we add DataSimulator.SubscribeBar = false; to the scenario code.

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework) : base(framework)
        {
        }

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

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

DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
DataSimulator.DateTime2 = new DateTime(2013, 12, 19);

DataSimulator.SubscribeBid = false;
DataSimulator.SubscribeAsk = false;
DataSimulator.SubscribeTrade = true;
DataSimulator.SubscribeBar = false;

//BarFactory.Add(instrument, BarType.Time, 60);
BarFactory.Add(instrument, BarType.Time, 300);

strategy.AddInstrument(instrument);

StartStrategy(StrategyMode.Backtest);

We run simulations and check MSFT data in the OpenQuant IDE. Now we see that we also have Bar.Time.300 series with 5 min bars.

We are ready for the final experiment in this chapter. Let’s disable trade simulations, comment out BarFactory.Add() and DataManager.Save() and uncomment bar playback from the local data series.

using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
Run simulations and you will see that you have 1min and 5min bars.

Now the question is what if we have several bar series in the local OQ database but we want to simulate particular bar series? What if we need to simulate only 5min bars in our exercise? The answer is in the
DataSimulator.BarFilter component. You can use the BarFilter to tell to the DataSimulator that we want to simulate bars of particular type and size. In our example we should add

```csharp
DataSimulator.BarFilter.Add(BarType.Time, 300);
```

code line:

```csharp
using System;
using SmartQuant;
namespace OpenQuant
{
    public partial class MyScenario : Scenario
    {
        public MyScenario(Framework framework) : base(framework)
        {
        }

        public override void Run()
        {
            strategy = new MyStrategy(framework, "Backtest");
            Initialize();
            Instrument instrument = InstrumentManager.Instruments["MSFT"];
            DataSimulator.DateTime1 = new DateTime(2013, 12, 18);
            DataSimulator.DateTime2 = new DateTime(2013, 12, 19);
            DataSimulator.SubscribeBid = false;
            DataSimulator.SubscribeAsk = false;
            DataSimulator.SubscribeTrade = false;
            DataSimulator.SubscribeBar = true;
            DataSimulator.BarFilter.Add(BarType.Time, 300);
            //BarFactory.Add(instrument, BarType.Time, 60);
            //BarFactory.Add(instrument, BarType.Time, 300);
            strategy.AddInstrument(instrument);
            StartStrategy(StrategyMode.Backtest);
        }
    }
}
```

And the output now
**Processing Data in the Data Simulator and using the DataProcessor to generate and emit BarOpen, bar and High, Low, Open and Close trades per bar.**

Before data objects get emitted from the DataSimulator into the EventBus they pass the DataProcessor. DataSimulator uses DataProcessor to filter, modify or emit new data objects.

The built-in DataProcessor allows doing several interesting and helpful things, namely if you have Bars in the simulated data series, the DataProcessor can generate and emit BarOpen event and also generate and emit four trades per bars, simulating High, Low, Open and Close trades.

Use default DataProcessor properties to disable or enable this feature

```
EmitBar (true)
EmitBarOpen (true)
EmitBarOpenTrade (false)
EmitBarHighTrade (false)
EmitBarLowTrade (false)
EmitBarCloseTrade (false)
```

By default the DataProcessor passes Bars through and generates additional BarOpen bar but doesn’t generate trades from incoming bar.

You can access DataProcessor using Framework.DataSimulator.Processor reference to change its properties or to assign your custom data processor to the DataSimulator.

The data processor has OnData method that you can override in your custom data processor to implement your own data processing and generation logic. Below you can find actual code of default DataProcessor, which you can use as a reference

The idea of this method is simple - return reference to a data object that you want to pass through, or use Emit(DataObject obj) method of the DataProcessor to emit new data objects. Return null if you want to filter incoming data object out.
protected virtual DataObject OnData(DataObject obj)
{
    if (obj == null)
    {
        Console.WriteLine("DataProcessor::OnData error. Data object is null.");
        return null;
    }
    if (obj.TypeId == DataObjectType.Bar)
    {
        Bar bar = (Bar)obj;
        if (emitBarOpen)
        {
            Emit(new Bar(bar.openDateTime, bar.type, bar.open, 0, 0, 0, 0, 0));
        }
        if (emitBarOpenTrade)
            Emit(new Trade(bar.openDateTime, 0, bar.instrumentId, bar.open, (int)(bar.volume / 4)));
        if (emitBarHighTrade && emitBarLowTrade)
        {
            if (bar.close > bar.open)
            {
                Emit(new Trade(new DateTime(bar.openDateTime.Ticks + (bar.CloseDateTime.Ticks - bar.openDateTime.Ticks) / 3), 0, bar.instrumentId, bar.close,
                             (int)(bar.volume / 4)));
                Emit(new Trade(new DateTime(bar.openDateTime.Ticks + (bar.CloseDateTime.Ticks - bar.openDateTime.Ticks) * 2 / 3), 0, bar.instrumentId, bar.close,
                             (int)(bar.volume / 4)));
            }
            else
            {
                Emit(new Trade(new DateTime(bar.openDateTime.Ticks + (bar.CloseDateTime.Ticks - bar.openDateTime.Ticks) / 3), 0, bar.instrumentId, bar.open,
                             (int)(bar.volume / 4)));
                Emit(new Trade(new DateTime(bar.openDateTime.Ticks + (bar.CloseDateTime.Ticks - bar.openDateTime.Ticks) * 2 / 3), 0, bar.instrumentId, bar.open,
                             (int)(bar.volume / 4)));
            }
        }
        if (emitBarCloseTrade)
            Emit(new Trade(bar.CloseDateTime, 0, bar.instrumentId, bar.close, (int)(bar.volume / 4)));
        if (!emitBar)
            return null;
    }
}
Strategy Trees

SmartQuant Framework provides several strategy classes and a possibility to combine strategies into hierarchical structures, namely into trees of strategies.

One strategy can become a parent strategy for another strategy. Consequently, the latter strategy becomes a child strategy for its parent strategy. A parent strategy can have several child strategies and each child strategy can act as a parent strategy for another strategy. This way a strategy tree can be formed.

The main feature of the parent–child strategy relationships and hierarchy is that the parent strategy portfolio aggregates portfolios of its child strategies. In other words if a portfolio transaction is added to a strategy, this transaction is also added to its parent strategy portfolio. If this parent strategy is in turn a child strategy for another strategy, this transaction is added to its parent strategy portfolio as well and so on up to the top level strategy portfolio. We can say that a transaction propagates from its original strategy portfolio to the top level strategy portfolio in the strategy tree. We can also say that strategy portfolios form a portfolio tree and the structure of this portfolio tree replicates the structure of the underlying strategy tree. Thus we can talk about strategy trees and portfolio trees in the same manner throughout this doc.

Creating a Tree of Strategies

Let’s develop an example. Assume we want to have a parent strategy that will control two child strategies. The parent strategies can for example re-allocate cash between child strategies, enable or disable child strategies depending on their performance and perform other high level risk and money management.

We derive root strategy class from ordinary Strategy class since we don’t need any sophistication offered by InstrumentStrategy or ComponentStrategy classes. Let’s call this class MetaStrategy since a root strategy looks and acts pretty much like MetaStrategy from OpenQuant 2013.

```csharp
using System;
using SmartQuant;

namespace OpenQuant
{
    public class MetaStrategy : Strategy
    {
        public MetaStrategy(Framework framework) : base(framework, "Meta")
        {
        }
    }
}
```
Two child strategies are going to be two instances of a simple BySellStrategy class derived from InstrumentStrategy class.

```csharp
using System;
using SmartQuant;

namespace OpenQuant
{
    public class BuySellStrategy : InstrumentStrategy
    {
        public BuySellStrategy(Framework framework, string name) : base(framework, name)
        {
        }
    }
}
```

**Controlling running strategies (changing parameters and calling methods)**

You can change parameters (fields or properties) or call methods of running strategies on the fly either directly in OpenQuant IDE or remotely using QuantController / QuantConsole.

Strategy properties that you want to change should be marked with `[Parameter]` attribute in the strategy code, for example in the case of Bollinger Bands demo strategy:

```csharp
[Parameter]
public double AllocationPerInstrument = 100000;

[Parameter]
public double Qty = 100;

[Parameter]
public int Length = 10;

[Parameter]
public double K = 2;
```

Once your parameters are marked with `[Parameter]` attribute and your strategy is running, you can open Strategy Manager Window in OpenQuant IDE to see the tree or running strategies.
You can click on a particular strategy and observe/change strategy parameters in the Properties Window.

- AllocationPerInstr: 100000
- K: 2
- Length: 10
- Qty: 100
Let’s change Qty parameter from 100 to 500 for AAPL strategy instance. We get the following message in the Output Window confirming that Qty parameter has been successfully changed.

Remember that when you run an InstrumentStrategy, you actually run a strategy tree with a parent strategy holding strategy instance per instrument. This is exactly what we see in the Strategy Parameters window – a tree of the BollingerBands instrument strategy with two strategy instances for AAPL and MSFT instruments.

Note that we can change parameters for a particular strategy instance / instrument. In our example we change Qty for AAPL instrument (Qty = 500), but it stays the same (Qty = 100) for MSFT instrument. If you want to change particular parameter for all instances simultaneously, you can select the parent strategy (BollingerBands root strategy in our case).

Let’s change Qty parameter to 250 for all instrument instances. The Output Window suggests that we have changed Qty parameter in the parent strategy and in both instrument instances, AAPL and MSFT.

Note that when you change strategy parameters, you only change values of corresponding fields or call setters for properties with explicit setters. This means that if you change K or Length parameters in our BollingerBands demo example, BBU/BBL indicators will not be actually affected since initial values of these parameters has already been passed to BBU/BBL indicator constructors at indicator creation time, and apart from this point, there is no connection between K or Length field and indicator objects. You
should do some extra work to pass new values of K and Length parameters to indicators if you really want your strategy to behave this way. On the other hand if you change Qty property, all new strategy orders will be send with this updated quantity.

You can also mark certain methods in your strategy class with [StrategyMethod] attribute. Note that such methods should take no parameters. An obvious example of such callable method is the Panic method of your strategy that usually closes all outstanding orders and halts future trading. In our tutorial we simply write Panic in this method to demonstrate how this concept works:

```csharp
[StrategyMethod]
public void Panic()
{
    Console.WriteLine("Panic " + Instrument);
}
```

Once you mark a method with [StrategyMethod] attribute and run your strategy, the name of this method will appear in the Strategy Monitor window under corresponding strategy folder:

If you double click on the method name, this method will be called in the corresponding strategy:
Similarly to parameter settings in the InstrumentStrategy, a method will be called for all strategy instrument instances if you double click on the parent strategy method name: