Introduction

To

SmartQuant Strategy Development

Version 2006-08-29



### 1 Introduction .........................................................................................................5

1.1 Goals of This Document ................................................................................................ 5

1.2 Intended Audience ......................................................................................................... 6

### 2 General Trading System Design Issues............................................................6

2.1 You and Your Trading System ...................................................................................... 7

2.1.1 Compatibility with Your Trading Beliefs.................................................................. 7

2.1.2 Use Technical Methods that You Believe In............................................................. 7

2.1.3 Testing Gives You Emotional Strength for the Hard Times...................................... 8

2.2 Major Types of Trading Systems................................................................................... 8

2.2.1 Systems Bet With or Against a Situation................................................................... 8

2.2.2 Systems are Long, Short, or Out................................................................................ 9

2.2.3 Breakout Systems ...................................................................................................... 9

2.2.4 Trend Systems ......................................................................................................... 10

2.2.5 Anti-Trend Systems................................................................................................. 10

2.2.6 Gap Closing Systems............................................................................................... 11

2.2.7 Spread Trading Systems .......................................................................................... 11

2.2.8 Volatility Systems.................................................................................................... 12

2.2.9 Pattern Matching Systems ....................................................................................... 13

2.2.10 Other Types of Systems ...................................................................................... 13

2.3 Money and Exposure Management.............................................................................. 13

2.3.1 Money Management................................................................................................ 14

2.3.2 Exposure Management ............................................................................................ 14

2.3.3 Diversification ......................................................................................................... 14

2.3.4 Stop Loss Orders ..................................................................................................... 15

2.4 General Strategy Pitfalls .............................................................................................. 15

2.5 Some Cautions on Assumptions .................................................................................. 16

### 3 SmartQuant Strategy Concepts.......................................................................17

3.1 Packaging follows Microsoft Visual Studio Models.................................................... 17

3.1.1 A SmartQuant Solution............................................................................................ 17

3.1.2 MetaStrategy Projects.............................................................................................. 18

3.1.3 Individual Strategy Projects..................................................................................... 19

3.2 Partitioned versus Integrated Frameworks................................................................... 21

3.3 The Integrated Framework (ATSMetaStrategy).......................................................... 22 3.4 The Partitioned Framework (MetaStrategy) ................................................................ 23 3.5 Strategy Components and MetaStrategy Components................................................. 24

3.6 Normal and Cross Components ................................................................................... 25

3.7 Unknown Components................................................................................................. 26

### 4 SmartQuant Integrated Framework Structure ................................................28

4.1 Section Outline............................................................................................................. 28

4.2 Using Daily Bars for Writing Convenience................................................................. 28 4.3 A Complete Code Example (Integrated Framework) .................................................. 29

4.4 Defining Variables, Properties, and Parameters .......................................................... 31

4.5 The Init Method ........................................................................................................... 32

4.6 Initialization................................................................................................................. 33

4.6.1 Defining Variables and Parameters ......................................................................... 33

4.6.2 Optimizing Parameters ............................................................................................ 34

4.7 ATS Component Properties, Methods, and Events...................................................... 34

4.8 Common Strategy Events............................................................................................. 36

4.8.1 OnBarOpen.............................................................................................................. 37

4.8.2 OnPositionOpened....................................................**Error! Bookmark not defined.**

4.8.3 OnValueChanged..................................................................................................... 37

4.8.4 OnPositionChanged................................................................................................. 38

4.8.5 OnBar....................................................................................................................... 38

4.8.6 OnBarSlice............................................................................................................... 38

4.9 Signals in the Integrated Framework ........................................................................... 39

4.10 How to Place Orders (Integrated Framework)............................................................. 39

### 5 SmartQuant Partitioned Framework Structure...............................................40

5.1 Components in the Partitioned Framework.................................................................. 40

5.1.1 The Main Goal of Components ............................................................................... 40

5.1.2 Control Flow among Components........................................................................... 41

5.2 Signals in the Partitioned Framework.......................................................................... 41

5.2.1 Signal Types ............................................................................................................ 42

5.2.2 Signal Flow among Components............................................................................. 42

5.3 Market Manager Component ....................................................................................... 43

5.4 Entry Component......................................................................................................... 44

5.4.1 How to Place Orders using Signals ......................................................................... 45

5.5 Exit Component ........................................................................................................... 45

5.6 Money Manager Component ....................................................................................... 46

5.6.1 Without Calling Risk Manager Helper Code........................................................... 46

5.6.2 With Calling Risk Manager Helper Code................................................................ 47

5.7 Risk Manager Component ........................................................................................... 48

5.7.1 Default Risk Manager Component.......................................................................... 48

5.7.2 A GetPositionRisk Example.................................................................................... 48

5.8 Exposure Manager Component.................................................................................... 49

5.9 MetaStrategy Components........................................................................................... 49

5.9.1 Simulation Component............................................................................................ 50 5.9.2 MetaStrategy Money Component............................................................................ 50 5.9.3 Metastrategy Risk Component ................................................................................ 51

5.9.4 MetaStrategy Exposure Component........................................................................ 52

### 6 SmartQuant Example Strategies......................................................................52

6.1 Summary of Strategy Techniques by Strategy............................................................. 52

6.1.1 5% Panic Recovery.................................................................................................. 53

6.1.2 Four Days Down and Long...................................................................................... 53 6.1.3 Four Days Up and Short for 1% Profit.................................................................... 53 6.1.4 Breakout with 4% Entry Limit ................................................................................ 53

6.1.5 Breakout with Multiple Exits................................................................................... 53 6.1.6 Bollinger Bands....................................................................................................... 53 6.1.7 Bollinger Bands with Profit Target.......................................................................... 53

6.1.8 Simple Moving Average Crossover......................................................................... 53

6.1.9 Slow Turtle Trend Following .................................................................................. 54

6.1.10 Chande’s 65sma\_3cc Crossover Strategy ........................................................... 54

6.1.11 Stock 2% Gap...................................................................................................... 54

6.1.12 Stock Down, Stock 2% Gap................................................................................ 54

6.1.13 QQQQ Gap Down 0.5%, Stock Down, Stock Gap 5%....................................... 54

6.1.14 QQQQ Gap 0.5%, Stock Down, Stock Gap 5%, Hold Overnight ...................... 54

6.1.15 Unilateral Pairs Trading ...................................................................................... 54

6.1.16 QQQ Crash, QQQQ Trade.................................................................................. 54

6.1.17 QQQ Crash, Stock Trade..................................................................................... 55

6.1.18 Bollinger Bands with 5-Minute Bars................................................................... 55

6.2 Pattern Matching Strategies ......................................................................................... 55

6.2.1 5% Down-In-One-Day Panic Recovery .................................................................. 55

6.2.2 Four Down Days and Long...................................................................................... 57

6.2.3 Four Up Days and Short for 1% Profit.................................................................... 58

6.3 Breakout Strategies...................................................................................................... 60

6.3.1 Breakout with 4% Entry Limit ................................................................................ 61 6.3.2 Breakout with Multiple Exits................................................................................... 62

6.3.3 Confirmation Methods............................................................................................. 67

6.4 Range Trading Strategies............................................................................................. 67

6.4.1 Bollinger Bands....................................................................................................... 68

6.4.2 Bollinger Bands with Profit Target.......................................................................... 70

6.5 Trend Following Strategies.......................................................................................... 73

6.5.1 Simple Moving Average Crossover......................................................................... 73

6.5.2 “Slow Turtle” Trend-Following .............................................................................. 77

6.5.3 Chande’s 65sma\_3cc Strategy................................................................................. 80

6.6 Gap Closing Strategies................................................................................................. 88

6.6.1 Stock 2% gap........................................................................................................... 88

6.6.2 Stock down, stock 2% gap....................................................................................... 91 6.6.3 Stock down, stock 5% gap....................................................................................... 92

6.6.4 QQQQ gap down 0.5%, stock down, stock gap 5%................................................ 93

6.6.5 QQQQ gap 0.5%, stock up, stock gap 5%............................................................... 95

6.6.6 QQQQ gap 0.5%, stock down, stock gap 5%, hold overnight................................. 97

6.7 Spread and Volatility Trading Strategies..................................................................... 99

6.7.1 Unilateral Pairs Trading........................................................................................... 99

6.7.2 QQQ Crash, QQQQ Trade .................................................................................... 104

6.7.3 QQQ Crash, Stock Trade....................................................................................... 106

6.7.4 Bollinger Bands with 5-Minute Bars..................................................................... 109

### Bibliography................................................................................................................112

### 7 Terminology.....................................................................................................112

# 1 Introduction

SmartQuant is a system for designing and executing computerized quantitative trading strategies. It is a very flexible and powerful system, so it contains many different software components and an associated degree of complexity.

Here is a list of other SmartQuant documents. If you’re a new SmartQuant user, you might consider reading the documentation in the following order to maximize your learning speed.

1. **System Architecture Manual**. This manual introduces SmartQuant system terminology, major system components, and system configurations for solving typical user goals.
2. **Getting Started Manual**. This manual shows you how to use the IDE to configure and execute a simple strategy on historical data stored in the IDE database. You can choose a financial instrument, run a strategy, and inspect the simulation results.
3. **Strategy Development Manual**. This manual shows you how to design and code user-defined strategies on user-specified financial instruments. The manual begins with an overview of trading system design, then moves on to SmartQuant system concepts, and finally discusses the SmartQuant code for several strategies in depth.
4. **Code Recipes Manual**. This manual contains code examples that show how to accomplish common user goals with the SmartQuant framework API.

## 1.1 Goals of This Document

This document is about how to write code for the SmartQuant system. It talks about the macro code structure of partitioned and integrated strategy frameworks (both of these terms are defined more clearly below). It talks about the micro code implementations of several example strategies. It talks about the code events that implementations can use to implement their strategies.

In contrast, this document does not talk about how to use the SmartQuant applications—the IDE, the DataCenter, and the CATS applications—to develop, back test, or live trade the strategies. See other manuals for how to use those programs to run the code strategies in this manual.

After reading this document, you should have a basic understanding of trading system design, including some critical things that you need to get right and some pitfalls that you need to avoid. You should also have a feel for the general theories behind popular types of strategies, and how to implement simple strategies—in code—within the SmartQuant system. Once you finish reading this document, it should be easy for you to write and back test your first few trading strategies.

In particular, this document discusses the following subject areas:

1. **General Trading System Design Issues.** Compatibility between you and your trading system; major types of trading systems; money and exposure management; diversification; general trading system pitfalls such as over fitting and brittle systems.
2. **SmartQuant Strategy Concepts.** Partitioned and integrated framework models; MetaStrategy-level versus individual strategy-level components; single-instrument versus cross-instrument components; unknown components in the IDE; signal flows in the partitioned framework.
3. **SmartQuant Integrated Framework Code Blocks.** A complete integrated framework code example; structure and content of C# code files; declarations; initialization method; system events (OnBarOpen, OnPositionOpened, OnBar, etc); code for placing trading orders; code for drawing a series on a price chart;
4. **SmartQuant Partitioned Framework Code Blocks.** Partitioning of code into partitioned components; Market Manager code; placing trading orders; Entry code; Exit code; Money Manager code; Risk Manager code; Exposure Manager code.
5. **SmartQuant Example Strategies.** Code for Breakout strategies; confirmation methods (e.g. 3 consecutive closes above a breakout point); oscillator strategies; moving average strategies; Gap closing strategies; Altucher strategies from *Trade Like a Hedge Fund* book; Chande strategy from the *Beyond Technical Analysis* book.
6. **Terminology.**  Definitions of terms used in this document; references to books on trading systems and strategy development.

## 1.2 Intended Audience

This document is an introductory document about how to write code for the SmartQuant system, so it only covers basic automated trading system concepts and some typical strategy scenarios. It does not cover coding techniques for advanced strategy development issues or data management.

# 2 General Trading System Design Issues

The goal of this section is to provide a perspective on the “big picture” of strategy development, before digging into the details of how to code strategies with the SmartQuant system. The big picture discussion talks about two important things:

1. Why it is important that you and your strategy are compatible, and what you can do about increasing the match between you and your trading system;
2. A list of some popular kinds of trading systems, so that you can pick a trading system that is compatible with your trading beliefs.

## 2.1 You and Your Trading System

Successful trading is not just about the trading system itself—instead, it’s about the *combination* of you *and* your trading system.

Many systems can provide good results according to their design. But that doesn’t mean that *you* can make money with them. Maybe you can’t stand the tedium of making too few or too many trades. Maybe you can’t stand the huge draw downs during hard times. Maybe you can’t let the trades run when they’re showing huge profits. All sorts of reasons and pressures can convince you to abandon a working trading system, and maybe to make it unprofitable for *you*.

So it’s not just about the trading system—it’s about your ability to trade it too.

#### 2.1.1 Compatibility with Your Trading Beliefs

The main compatibility problem is that in the heat of trading, you might want to do something different than the trading system wants to do. Who gives the orders?

For example, if you’re in a deep drawdown with fully 50 percent—half!—of your equity gone into losing trades, what do you do when the trading system is telling you to keep on riding the existing (and future) trades that the system is thinking up? Your risk control alarms are screaming to close the positions, because the “portfolio heat” is too much for you. What do you end up doing? How can you mitigate or avoid this problem?

The main solution to the compatibility problem is to trade a system that you believe in, so that you can stick with it during hard times. You can’t do anything about the market, but you can do something about yourself and your trading system. Pick an approach that you believe in, and then test it extensively to deepen your understanding and belief in it. Run it with reasonable money management (trade sizing) and exposure (portfolio risk) constraints. Only then will you have a chance of sticking with the system when your emotions run high and your equity runs low.

#### 2.1.2 Use Technical Methods that You Believe In

To increase your chances of trading success, pick a trading approach that you believe in. There are many types of profitable systems to choose from—here is a list of some of them:

* Breakout Systems
* Trend Following Systems
* Trend/AntiTrend (Reversal) Systems
* Range Trading Systems
* Gap Closing Systems
* Volatility Systems
* Intermarket Correlation Systems
* Arbitrage Systems

More information on these trading systems is provided in the sections below. Read the general description of each system, and see which ones intuitively appeal to you the most. Perhaps those systems might be good starting points for your next strategy development project.

You might also consider the list of book references near the beginning of this document for more ideas and deeper information on trading systems of various kinds. Many of the books in the list are excellent sources of useful trading information.

#### 2.1.3 Testing Gives You Emotional Strength for the Hard Times

Once you’ve learned enough about trading systems to pick one that you like, the most important thing you can do to deepen your belief in the system is to back test it against various sets of historical data for various markets.

If you do a good job of testing, so that you *know* that your system is robust enough to survive (and be profitable) in a wide variety of past market situations, then you will have more belief in it’s behavior and abilities in future markets, during horrendous draw downs.

In particular, once you are satisfied with your back testing results on historical data, the next step forward is to “paper trade” your strategy against real time data (but with fake money). This is an important step to take because it exposes your strategy (and you) to the ups and downs of the real market, in real time. Once you are satisfied with your paper trading results for a few days or weeks or months (the interval must fit your emotional needs), then you can start live trading your strategy and feel confident about your expected results.

## 2.2 Major Types of Trading Systems

This section summarizes some popular kinds of trading systems—what concepts they’re based on; what they’re good at; and what they’re not good at. These summaries are not comprehensive. They are only intended to draw a “big picture” landscape of possible trading system types, from which you can choose particular strategies that interest you. All of these strategies are explained in the book reference list near the beginning of this document.

#### 2.2.1 Systems Bet With or Against a Situation

To help novices characterize trading systems, it is worth mentioning that there are two general classes of trading systems—those that expect a situation to continue (to trend forward), and those that expect (or try to force by trading) a situation to change.

Systems that bet with a situation are all trend following systems of one kind or another. They use different technical mechanisms to enter and follow and exit a trade, but they all go with the initial change, and hope it becomes a trend that they can follow for profit. Two examples of these systems include the Breakout and Trend following systems described below.

In contrast, systems that bet against a situation include many systems that are qualitatively different from each other. Some systems bet against simple price direction (Anti-trend systems). Some bet against open gaps (Gap Closing systems). Some bet against differences in prices for the same thing in different markets (Arbitrage systems). And some bet against the average value of something (Mean Reversion systems), where the average value can be anything from the direction or distance between two price series to the volatility of a single price series or of the distance between two series.

Furthermore, systems that bet with a situation (such as trend following systems) usually want the situation to continue for a long and profitable time, whereas systems that bet against a situation want the situation to change as soon as possible (so as to capture a profit from the change as quickly as possible). So if the markets permit it, and if the strategy entry criteria permit it, “bet with” systems will make fewer and longer trades to earn their profit, while “bet against” systems will usually try to make more trades and shorter trades to earn their profit. (The trade entry criteria are important—for example, your gap-closing strategy could wait for months before it sees a 5% gap down to bet against).

#### 2.2.2 Systems are Long, Short, or Out

Systems can also be classified by their possible trading states—in or out of the market (state), and (direction) long or short the market.

Single state systems are always in the market. Single state systems are not selective about market conditions. They can only decide to be long or short, even if market conditions do not favor any trades at all (such as a very flat market).

Two-state systems can be in or out of the market. These systems can be patient and selective about which trades they want to enter. They have the option of sitting out unfavorable market conditions, so they are more flexible than single state (always in the market) systems.

A unidirectional system can go long, or short, but is restricted so that it cannot switch between long or short. For example, a conservative system might only allow long trades. In contrast, a bidirectional system can go long or short as it pleases.

Note that single state unidirectional systems make no sense at all—they would have to be permanently in the market long or short forever. So for practical purposes, all single state systems must be bidirectional (like a trend-reversal or trend-flipping system).

#### 2.2.3 Breakout Systems

The concept of a breakout system is that when something significant happens to an instrument, the price will break out (either up or down) of its previous trading range. Breakout systems monitor the usual width of the price channel (the range), and open new positions when the price breaks out of the channel.

One of the oldest futures trading systems was based on this idea. The typical trading rules were something like: Buy when the price breaks above the highest high of the past four weeks, or below the lowest low of the past four weeks. Sell when you reach a profit target, or when a trailing stop loss was hit.

Breakout systems are good for capturing moves that break out and take off some distance from the channel, and they save you from trying to trade small moves as prices whipsaw back and forth within the price channel. But they can whipsaw you too, by providing false breakouts that suddenly return back into the channel after you’ve open a position in the breakout direction.

Some breakout systems use extra criteria to “confirm” that the breakout is valid. For example, you might require that one, two, or three bars close outside the channel before opening a position. Further, you might require that all three bars close farther away from the channel each day.

In all strategies, adding more selective confirmation criteria will usually decrease the number of trades made because more trades (both winning and losing) are excluded. But if the criteria ideas work well, both the accuracy (% winning trades) and profitability of the system will increase.

#### 2.2.4 Trend Systems

The concept of a trend following system is to open a position in the direction of the current trend, and then keep the position open until the trend reverses. Usually trend systems are constructed from a pair of moving averages that produce trading signals when they cross over each other. One moving average is faster (fewer days in the length of the average) than the other, so it hugs the price movement more closely. The other average is slower (more days, more length), and so takes a different “path” through the price chart than the fast average does.

A typical set of trading rules is like this: Buy when the fast average crosses above the slow average, and sell when the fast average crosses below the slow average. (If you were trading both long and short sides with the system, you would open a short position here too, and reverse it back to long at the next crossover point.)

Trend following systems can generate huge profits on long trends that run for months or years, if the system doesn’t exit prematurely on price pullbacks that force the moving averages to cross and exit the trade. There is somewhat of an art to choosing the best types of averages for a particular market—simple, exponential, or weighted—and lengths of averages, to allow enough pullback room so your position is not taken out on small price moves in a longer trend.

The biggest problems with trend systems is that they also give up a significant amount of money at tops and bottoms while waiting for the averages to cross over, and they can whipsaw you into heavy losses in a flat, range-trading channel kind of market as they force you into unprofitable trades when there is not enough price movement between crossover points.

#### 2.2.5 Anti-Trend Systems

The concept of an anti-trend system is to open a position *against* the direction of the trend, in the expectation that the trend will reverse enough to make the position profitable. A key point of this concept is that only *short term* trends are considered, because short term trends often do reverse, and short term anti-trend trades often do earn a profit. Of course, if the trend doesn’t reverse soon enough, or if it continues too far in the original trend direction, the position is closed at a loss.

Many different technical mechanisms can be used to construct anti-trend systems. But generally speaking, a moving average of some kind is used to represent the center of the trading channel. This might be an 18 or 30 day moving average, for example. In addition, two other lines, bands, or boundaries are used to represent the outer range of the average trading channel. When prices move far enough away from the average toward one of the outer channel boundaries, a trade signal is triggered.

A typical set of trading rules might be: Buy when prices trend down below the moving average by more than 1 standard deviation of the normal trading range. Sell when prices return to the moving average, or when a profit target is reached. Also sell if the trend continues on too far in the original direction, or if the trade is still open 1 or 2 bars later.

Anti-trend systems are good for flat “sideways” markets that have enough movement in them to make anti-trend trades worthwhile. But be aware, anti-trend systems are susceptible to real trends that keep on going, forcing you to close positions at a loss.

#### 2.2.6 Gap Closing Systems

The concept of a gap closing system is that gapping prices will retreat back to close the gap sooner or later. By opening a position against the movement that caused the gap, you can make a profit when prices return to fill the gap. Both overnight and intraday gaps are tradable.

A typical set of trading rules might be: Buy when prices gap down more than 2% below the previous bar close. Sell when prices rise to the previous close, or sell at the end of the trading day. Additional confirmation conditions might be to require the previous day to be a down day, or a down day for both the market and the stock.

Gap closing systems are very popular with many kinds of traders. That’s because traders can make profits on gaps that are caused by news events or strong (but short term) price pressures. However, gaps can lose traders money too, if gaps are caused by legitimate price pressures that force prices to continue on in the original trend. Worse yet, because gap pressures are strong, prices can move fast against you and generate losses for you quite quickly if you’re not careful.

#### 2.2.7 Spread Trading Systems

The concept of a spread trading system is to trade the spread between any two things that normally travel together. When the two things get too far away from their normal relationship, you open a position that anticipates their return to normal conditions. Spread trading systems usually have a structure that includes a spread between two instruments or price series.

For example, you might make a trade that says that the Dow Jones index normally trends in the same direction with the S&P500 index, or that two currencies from adjacent countries travel closely together. If the two get too far apart, you would buy one and sell the other, anticipating a return to normal separation distance at a later time (hopefully soon).

A typical set of trading rules might be: Buy the Nasdaq 100 (QQQQ) and sell the S&P500 (SPY) when the ratio between the QQQQ and SPY is more than 1.5 standard deviations below the normal 10-day moving average value of the ratio, and if the QQQQ has had a big -2% down day the day before. Close the position when the ratio moves back to the 10-day moving average.

To be clear, 1.5 standard deviations equates to 90% of a Normal Distribution, so a move that is greater than 1.5 standard deviations from the mean is (in theory) supposed to be greater than 90% of all the moves that the ratio has ever made. So that kind of a move doesn’t happen very often.

Spread trading systems are popular. You can spread trade (or ratio trade) any two things that are correlated closely enough to travel together a large percentage of the time. For example, two closely competitive stocks, two closely related currencies, or two closely related indexes can be used to trade this kind of strategy.

Notice that to neutrally trade a spread between two things, you must buy one and sell the other simultaneously. That’s the only way to isolate the spread between two moving price streams. Then if you enter the trade because the spread is too wide, you don’t care if both prices rise or fall while you’re waiting for the spread to close. Gains or losses in one instrument will be almost perfectly offset by corresponding losses or gains in the other instrument. All you care about is whether the spread between them gets narrower or wider.

If you want to take a ratio between two price streams, and then trade standard deviations on the ratio using only one instrument, that’s a volatility trade, which is not neutral to rising or falling prices in the two underlying instruments. Volatility trades are discussed below.

#### 2.2.8 Volatility Systems

The concept of a volatility system is to trade the amount of volatility that occurs in a single series of instrument prices. When you trade volatility, you’re actually trading the volatility of a thing against itself (usually against a moving average of itself).

One example of trading volatility is to trade the Nasdaq 100 (QQQQ) prices against a moving average of those prices. Another example—that was made famous by the Long Term Capital Management story—is to trade the implied volatility in equity options prices against the historical averages of the volatility. (The market was pricing in roughly 20% implied volatility, but the historical average was more like 15%, so LTCM shorted option prices, anticipating closing their positions when volatility—and therefore option prices—returned to the historical average. The strategy was sound, but during their trading timeframe, the volatility kept getting worse instead of better, eventually losing $4B in 4 weeks, and forcing them to close their company.)

A typical set of trading rules might be: Buy QQQQ when the price drops more than 1.5 standard deviations below the 10-day moving average of QQQQ. Sell QQQQ when the prices return back to the average, when a profit target has been reached, when a stop loss has been reached, or when a month of trading days has passed. (Remember that 1.5 standard deviations equates to 90% of a normal distribution, so a move that is greater than 1.5 standard deviations from the mean is supposed to be greater than 90% of all the volatility price moves that QQQQ has ever made.)

Notice that while the trade above is in effect, profits and losses are subject to rising and falling prices of QQQQ. The current QQQQ price might be 2 or 3 standard deviations below the moving average of QQQQ (enabling you to enter the trade long on QQQQ), but you will still lose money if QQQQ keeps dropping like a stone for a few days before the moving average catches up and the current price moves back to within 1.5 standard deviations of the moving average.

Volatility systems are popular with many traders, because tradable volatility is always present in the market somewhere. It works best with instruments or series that have a very low probability of maintaining high volatility for long periods of time (or worse yet, of increasing volatility so that 2.0 standard deviation events (>94% of all moves) are much more common than usual.)

Volatility trades do not work well with series that have “fat tails” on their distribution curves, because event distributions that have fat tails have many more events out there on the tails of the distribution curve (>1.5 or >2.0 standard deviations from the mean) than theory would predict. And since fat tails can cause losses (massive in the case of LTCM), you should try to allow for fat tails as much as is feasible when you’re trading volatility.

#### 2.2.9 Pattern Matching Systems

The concept of pattern matching systems is to recognize a particular bar pattern in market data and then make a trade that anticipates a future (profitable) situation that is suggested by the pattern. For example, a pattern might try to recognize double bottom chart patterns, or head and shoulder chart patterns, or other technical chart patterns (with or without the help of technical indicators such as moving averages, RSI indexes, or oscillators of various kinds).

A typical set of trading rules might be: After a trend upward of 15%, if you see a double-top chart formation, sell after the trailing edge of the double top breaks through the price level of the middle valley in the pattern.

These kinds of systems can be profitable (1) if the market serves up the desired chart patterns with enough frequency, (2) if the system can spot the pattern with enough accuracy, (3) if the pattern accurately predicts a future situation that is tradable, and (4) if the system can actually execute the trade profitably. As you might guess, this is a tough game to win.

#### 2.2.10 Other Types of Systems

There are many other types of trading systems that were not discussed above. For example, intermarket systems trade instruments from multiple markets against each other, using either positive or negative price correlations or movements to enter, size, and exit trades. Cause-and-effect systems use movements in one instrument (e.g. bond prices) to anticipate movements in other instruments (e.g. S&P500 stock indexes) that are supposedly related by weak or strong causeeffect relationships. These systems are all beyond the scope of this introductory document.

## 2.3 Money and Exposure Management

This section talks about how you can limit the damage that your trading system can do to your equity curve (recall that *you* are a big part of the potential damage sources). Essentially, there are only two ways to limit risk and damage—restrict the amount of equity used (per trade, per sector, or per portfolio), and diversify among uncorrelated investments.

You might restrict the amount of equity used in *new* trading positions, so that when a losing trade is made, the loss on that trade is limited to the amount of the original equity put into the trade. A variation on limiting equity loss for a particular trade is to place a stop loss order on the trade, to stop losses from growing beyond a particular limit.

You might also restrict the amount of equity used in *existing* positions that get too big, especially when the market value of positions changes enough to violate a risk limitation rule. Such equity restrictions would initiate trades to reduce or correct the equity position to bring it back within limits. For example, if your technology investments increase in value so much that they exceed your rule of “no more than 20% invested in technology”, then your strategy would initiate trades to sell some technology investments to bring the portfolio total within limits.

Finally, you might diversify the use of equity in uncorrelated ways, so that not all your trades or positions are likely to lose at the same time. With reasonable diversification in place, it is more likely that you should have some winners to offset the losers, thereby reducing equity volatility.

#### 2.3.1 Risk and Money Management

Money management is usually seen as the activity of sizing trades according to various monetary limits. For example, a typical money management rule might be “use 2% of equity per trade.”

Money management tends to operate at the scope of individual trades, rather than at higher levels that contemplate multiple instruments, multiple strategies, or multiple trading systems.

There are many ways to limit equity used—to name a few—restrict the equity used per trade, per instrument, per asset class, per strategy, per market, per portfolio, or per trading system. The main idea is to compartmentalize your equity, so that a single losing event in a single compartment (trade, instrument, strategy, or whatever) can only destroy the equity in its own compartment, instead of destroying the equity in all compartments.

Money manager software components can perform risk management calculations on their own, in order to calculate the size of pending trades. But it is also quite common for Money components to call risk manager helper components to do the risk calculations, specify trade size limits, or to watch existing positions and set stop loss orders when position values go outside of limits. Risk calculations can look at whatever information is needed for the desired calculation—trade conditions, portfolio values, and rules of all kinds.

#### 2.3.2 Exposure Management

Exposure management is similar to money management, but it operates at a higher level instead of at the trade sizing level. Exposure management looks at overall portfolios, to see if all existing portfolio positions are within exposure management rules and limits. For example, a typical exposure management rule might say “Limit technology to 20% of portfolio equity maximum.”

For example, if portfolio exposure is too great in a particular area such as technology, an exposure manager would at least block incoming trades that sought to increase portfolio exposure to technology stocks even more. In addition, the exposure manager might also initiate trades to *reduce* (correct) the exposure to technology in order to bring it back within the exposure limits.

Exposure manager software components can perform risk management calculations in order to carry out their responsibilities. The risk calculations can look at whatever information is needed for the desired calculation—trade conditions, portfolio values, and rules of all kinds. The key point of this paragraph is that risk calculations support the activity of exposure management. That is, mere risk calculations are not a separate “risk management” function by themselves.

#### 2.3.3 Diversification

Once you have split your equity into compartments, you should try to diversify the compartments among *uncorrelated or negatively correlated* trades, instruments, strategies, and systems. Diversification is based on the idea that there is a lower probability of simultaneous losing events in all of your *uncorrelated or negatively correlated* compartments of equity. Diversification (like money and exposure management) cannot change risk factors, but it can spread them around into separate compartments to reduce the probability of simultaneous failure in all compartments.

It is important to try to diversify using things that are uncorrelated (or at least, not highly correlated), because you want them to behave as independently as possible. That way, they are less likely to both go up or down at the same time.

Here are some poor examples of diversification, because the elements in these pairs normally travel together (high positive correlation): stocks of two disk drive manufacturers; two trend following systems with similar behavior; futures instruments for two closely related currencies; or exchange traded funds for the Dow Jones (DIA), the S&P500 (SPY), and the Nasdaq (QQQQ).

Here are some better pairs that show lower (or negative) correlation: a real estate stock and an industrial stock (real estate usually has a low correlation with the stock market); gold and bonds (when inflation goes up, gold goes up and bonds go down); a high-tech growth stock and a defensive “refrigerator” stock (when the economy goes down, the growth stock goes down, but people still have to buy food, so the defensive stock goes up); a trend following system and a volatility trading system (one catches trends, one catches sideways volatility).

For example, if you diversify your equity into a trend following system compartment and a volatility trading system compartment, the odds are more favorable that your equity growth curve will be less volatile. This is because the trend system is more likely to earn money in a trending market, and the volatility system is more likely to earn money in a choppy market. So the systems will tend to offset each other, reducing volatility in your equity curve.

Pay close attention to money and exposure management, because trading systems are not perfect. Compartmentalize and diversify your equity for better safety and smoother performance.

#### 2.3.4 Stop Loss Orders

It is important to consider using stop loss orders on all your automated trades, to prevent large losses in cases where the trade goes against you. Although some kinds of trades will automatically limit your maximum losses to the original trade equity (e.g. long stocks, long calls and puts), many kinds of trades can expose you to theoretically unlimited losses (e.g. short stocks, uncovered short calls, futures).

So to avoid losing the entire equity in a trade, you should consider entering stop loss orders as soon as you open a new position. You will see later that the event handler OnPositionOpened is a good place to automatically enter stop loss orders, because that event is fired whenever a new position is actually opened after your strategy gets a confirmation from your broker.

## 2.4 General Strategy Pitfalls

The very first pitfall to avoid is to use bad data for development and back testing (“Garbage in, garbage out.”) So try to use reasonable data. Hopefully your data will be good, and your strategy will continue to work when you start paper trading (with live market data).

Another common pitfall is to over optimize a system to fit a particular set of historical market data, so much so that the system will not work well on new live market data. This condition is called “over fitting” or “over optimizing”. One way to counteract over fitting is to limit the number of tunable optimization parameters in your system, and to deliberately choose reasonable values manually instead of optimizing them to multiple decimal places.

A second way to avoid over optimize a strategy is to use “out of sample” data to validate your strategy after you develop the strategy using “in sample” data. The main idea here is to develop your strategy on one set of historical data, and then validate its behavior on an equivalent set of historical data that the strategy has never seen before. If the strategy still behaves well, it is *not* because you have tuned the strategy to the out of sample data, because the strategy was not developed using the out of sample data.

Another common pitfall is to build a brittle system—one whose performance “breaks” (varies widely in profitability) with minor changes in system parameter values. To detect brittle systems, vary each system parameter during testing to see how sensitive system performance is to minor variations in parameter values. If the system is brittle, you should try to modify the system to be less brittle and more robust (insensitive to minor parameter changes). If you don’t, system performance on new market data is likely to “break” easily, and become unprofitable.

Still another pitfall is to build complex systems that have too many trading rules, too many parameters to coordinate and control, or too many sensitivities to unpredictable market price variations. Systems like this can exclude (or execute) too many good (or bad) trades, leave trades earlier or later than desired, and can produce wild swings in equity because they make too many unprofitable trades. Many experienced system designers have the same advice—keep your systems as simple to maximize your chance of building a workable, tradable, robust system.

Yet another pitfall is to not consider all transaction costs when back testing. Your strategy might earn a small profit on each trade, but it is possible that the profits will be eaten up (and more) by price slippages, commissions, and other account costs such as margin interest costs. So be sure to model transaction costs during your back tests so that you have an idea of whether they are significant or not for your particular strategy. Paper trading your strategy with a live trading account can help to expose some of these “hidden” costs for you.

A final pitfall is to think that your strategy should behave very well in all market conditions—in strong trending, strong ranging, and high and low volatility markets. It is extremely difficult to build a computerized strategy that is stable and reasonably profitable in all kinds of markets and all kinds of market conditions.

## 2.5 Some Cautions on Assumptions

Now that we’ve talked about some general system design issues, it seems useful to caution readers against some intuitive assumptions about trading systems that are easy to make. To counteract these intuitive assumptions, we mention a few of them here explicitly.

First, for trading systems in a real market, selling short is not the exact opposite of buying long. Some strategies just don’t work well on the short side, even though they work fine on the long side. Perhaps it’s because of an upward bias in markets, or more bullish market players than bearish market players. No one knows why for sure. But be aware, you cannot always flip a long strategy to the short side and expect to make comparable profits.

Second, what works for one kind of futures market doesn’t always work for stocks, bonds, indexes, or all other instruments (and vice versa). Technical analysis purists might argue that a price bar series is no more than a series of numbers, and that a good trading system will work on any series. And certainly much experience indicates that some trading systems will in fact work well on 10 or 20 different commodities or index futures markets, even including stock indexes.

And even some stock trading systems will work on futures markets. But be aware, you won’t be able to make this claim for *your* system unless you test it thoroughly.

Third, miscellaneous financial charges such as commissions, bid/ask spreads, liquidity effects, implied volatilities effects (for options), and trading slippage can all add up to significant amounts of money. Especially if your trading system has a low expected value per trade, these kinds of financial charges can make the difference between a profitable, tradable system and one that loses money. So be aware, your back testing runs should provide enough allowance for commissions, bid/ask spreads, and slippage effects to make the testing (and therefore the results) as realistic as feasible. Remember that when you test your system, you’re trying to increase the depth of belief in your system—so knowing that you have not allowed for slippage costs in your results will act to reduce your belief in your system when you will need it the most (probably in your worst time of draw down need).

# 3 SmartQuant Strategy Concepts

This section marks the transition from general information about trading systems to specific information about the SmartQuant trading system framework. From here on, we progress from high-level information about strategy structures to low-level information about how to code specific trading strategies.

We begin by talking about the way SmartQuant (1) packages and (2) implements strategies.

## 3.1 Packaging Strategies in *Solutions* and *Projects*

SmartQuant packages strategy *frameworks* in *projects* that follow the model used by Microsoft Visual studio. To be clear, **you cannot run a strategy directly in Visual Studio**. This is because strategies need to run on top of the whole underlying SmartQuant framework system, which is not part of Visual Studio.

Here are some pictures and definitions that will help to explain the following discussion on SmartQuant strategy packaging and strategy concepts.

#### 3.1.1 A SmartQuant Solution

A SmartQuant *solution* is a form of *packaging.* When you create a blank *solution* in SmartQuant, you get a top-level solution file that has no packaging metastrategy projects and no strategy component contents, as the following screenshot shows.



#### 3.1.2 MetaStrategy Projects

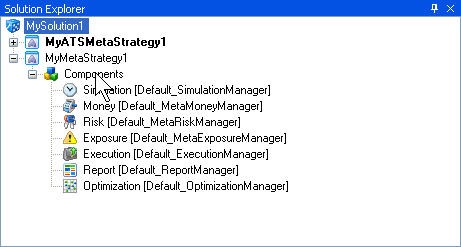
Once you have created a blank (packaging) solution, the next step is to create one or more metastrategy (packaging) *projects.* Two kinds of metastrategy projects are provided—one is called a *MetaStrategy project*, and the other is called an *ATSMetaStrategy project*.

Here is a screenshot that shows the *solution* from above, with two metastrategy *projects* created inside the solution. One metastrategy project of each kind has been created in the solution, to show how they are both located at the same conceptual level in the SmartQuant strategy “packaging” hierarchy. (To create a new metastrategy in the solution file, right click it.)



Metastrategy projects contain *components* (functional software files) from a *framework*, so metastrategy projects are not “empty” of content like the solution files are. First we’ll show what the components of a metastrategy look like in the IDE, and then talk about what they do.

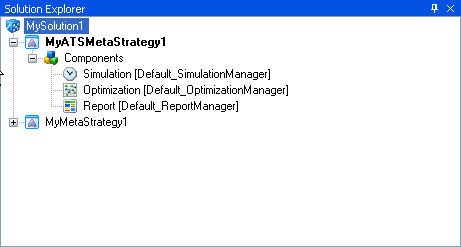
Here is a screenshot of the components in a MetaStrategy project. These components are part of a *partitioned strategy framework* (MetaStrategy projects use a partitioned framework model for implementing a strategy.)



Metastrategy components operate at the metastrategy level, which is a level above any particular strategy. Accordingly, metastrategy components typically provide general money, risk, exposure, and execution support across all individual strategies that are contained within the MetaStrategy project. (A metastrategy project can contain multiple strategies, such as one breakout strategy, one trend following strategy, etc.)

For example, suppose you have a trading rule that no strategy can allocate more than 5% of trading equity to technology stocks, and another rule that says no more than 10% of the portfolio can be allocated to technology stocks. Also suppose that you have 3 individual strategies (breakout, trend following, volatility) running within the same metastrategy project. Each individual strategy can enforce its own 5% rule, but only the metastrategy level components can enforce the “10% of total portfolio” rule (or other higher level rules that apply across strategies). This is because individual strategies cannot “see” into what other individual strategies are doing—only the metastrategy components at the metastrategy can do that.

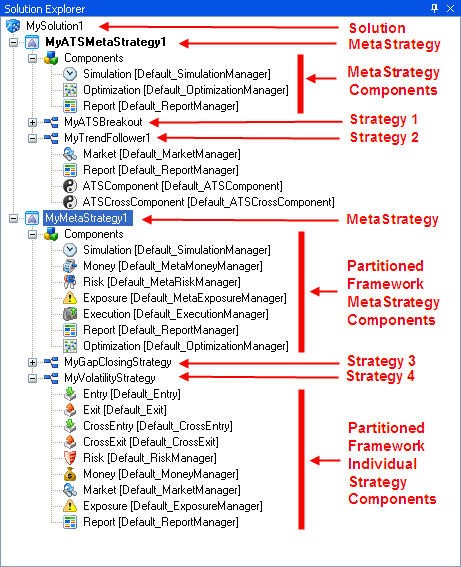
Here is a screenshot of the components inside an ATSMetaStrategy project. These components are part of an *integrated strategy framework* (ATSMetaStrategy projects use an integrated framework model for implementing a strategy.) As before, these components operate at a level above all the individual strategies that might be contained within the ATSMetastrategy project.



#### 3.1.3 Individual Strategy Projects

Once you have created a new solution project and have added one or more metastrategy projects, you are ready to add one or more individual strategy projects to the metastrategy projects. The individual strategy projects contain components that contain most of the code for your favorite computerized trading strategies.

Here is a screenshot that shows several individual strategy projects underneath the metastrategy projects that were shown above. The screenshot shows a breakout strategy, a trend following strategy, a volatility strategy, and a gap closing strategy. The individual strategy projects are implemented with the appropriate type of framework (partitioned or integrated) for their metastrategy project types. For clarity and review, the various types of solutions, metastrategy projects, strategy projects, and framework components have been labeled in red.



The top level *solution* contains no functional code components, but it contains metastrategy projects of two types (MetaStrategy and ATSMetaStrategy). The solution can also contain *Code Projects* (not shown) that are normal C# code projects that you can code any way you want. Code projects are also sometimes called “research projects”, because they are useful for testing out little bits of new C# code outside of a SmartQuant metastrategy project.

The two types of metastrategy *projects* (MetaStrategy and ATSMetaStrategy) both contain code components that operate at the *metastrategy level*, above any *individual strategies*. Typically components at the metastrategy level provide functions that apply to all individual strategies in the metastrategy project.

**BEWARE—by default, the IDE creates strategy projects that do not work on the example data that is shipped with SmartQuant. This is because newly created metastrategy projects use only Default\_XXX (that is, empty) strategy components that do nothing.**

Thus to get a new strategy project to work, you must go through the component types and select working components. This process is explained in more detail below, where this warning is given to you again. (This warning bears repeating because it can save you more than a few hours of time debugging working—but empty—“do nothing” strategy projects).

Now you have seen the structure of the SmartQuant “packaging” hierarchy. Next, we look at *strategy frameworks*, which contain the bulk of the code for your individual strategies (such as breakout, trend following, volatility, etc.) There are two kinds of strategy frameworks to consider—(1) a *partitioned framework* that uses separate components for trading functions (Entry, Exit, Money, Risk, Exposure), and (2) an *integrated framework* that packs all trading functionality into one big component (ATSComponent). We’ll look at both types.

## 3.2 Partitioned versus Integrated Frameworks

The SmartQuant IDE provides two strategy frameworks to make your development life easier. The two strategy frameworks are organized around two dimensions—(1) whether you want to partition your trading system into many separate functional components (Entry, Exit, Risk, Market) or group them into one big component that does everything (ATSComponent), and (2) the number of instruments that a strategy wants to “see” within each framework component.

The two types of frameworks are the *Partitioned* and *Integrated* frameworks, also known as the MetaStrategy and ATSMetastrategy frameworks. But “MetaStrategy” and “ATSMetastrategy” are so similar in form that they are easy to confuse. So we will not use them in this document any more than necessary. Instead we will use the terms *Partitioned* and *Integrated*, because these terms better express the essential difference between the two framework models.

The partitioned framework implements a strategy project using separate components (files) for the main functional roles in the strategy. Accordingly, the partitioned framework has separate components (files) for each strategy-level function such as Market Manager, Entry, Exit, Money Manager, Risk Manager, and Exposure Manager. In addition, the partitioned framework also provides a second set of components at the metastrategy level (Simulation, Money, Risk, Exposure, Execution, etc.)

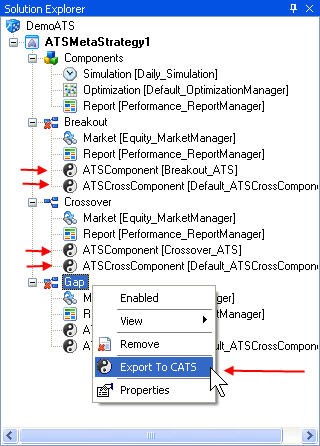
The main purpose of the partitioned framework is to make it easy for strategy developers to mix and match different components easily, for fast comparison of combinations of various trading algorithms in different components. For example, in a partitioned framework, you can easily right click a component in the Solution Explorer panel to switch a component to a completely different algorithm. Click again to run a back test with the new mix of components. The partitioned framework is intended for easy exploration and testing of new ideas. It is not intended for complex strategies that require lots of data sharing among components. You should use the integrated framework for complex cases like that.

In contrast, the main purpose of the integrated framework is to make it easier for strategy developers to do more complex things in their strategy projects. For example, the integrated framework makes it easier to share data among the code for different strategy functions, and to work with multiple instruments within any part of the integrated framework. The integrated framework is intended for professional trading systems, and is the only framework that can be used with the CATS system. So if you are a intending to write a complex, industrial-strength CATS trading strategy, be sure to use integrated frameworks for your projects.

## 3.3 The Integrated Framework (ATSMetaStrategy)

The most complex trading systems should use an integrated framework project as the foundation of their trading system. This is because the top-level ATSComponent (inside the ATSMetaStrategy project) combines all the Entry, Exit, Money, Risk, and Market components into one single component, so that your code can access all aspects of your strategy at all times.

Here is a screenshot showing the organization of the example DemoATS project. This ATS project has 3 strategies in it, plus the top-level section for the entire trading system. Notice that each of the three strategies (Breakout, Crossover, Gap) has three mandatory components (Market, ATSComponent, and ATSCrossComponent). The Report component is not mandatory—the default component is good enough for most strategy development situations.



Typically you would only put your strategy code in one of the two ATS components, and leave the other component blank by selecting the default version of it. Here, normal ATS components carry the code, and the ATSCrossComponents are empty.

When you right click a strategy node (as shown above for the Gap strategy), you get a context menu that lets you enable or disable the execution of the strategy within the metastrategy project, plus an option to export the strategy to a precompiled assembly for use in the CATS system.

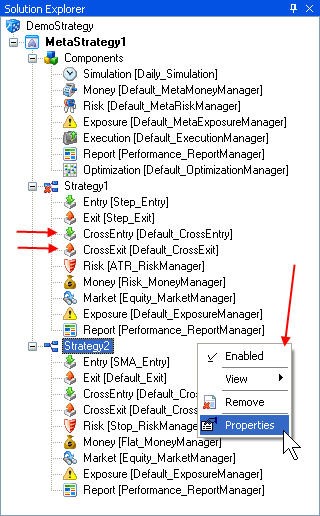
## 3.4 The Partitioned Framework (MetaStrategy)

In contrast, if you prefer a more structured and partitioned development approach, you should use a MetaStrategy project as the foundation of your trading system. A MetaStrategy project partitions various strategy functions into separate individual Entry, Exit, Risk, Money, Exposure, and Market components.

The main advantage of using a component approach is that you can easily reconfigure your strategy to try out different combinations of components. This “mix and match” capability is possible because all components of a particular type (Entry, Exit) have the same API, and are interchangeable. Just right click the component type name in the Solution Explorer panel of the IDE, and choose a component instance (or combination) that interests you.

Metastrategy projects (both types) allow you to create multiple individual strategy components within one project. This means that you can run multiple simultaneous different strategies against the same data stream, and collect aggregated performance statistics on the set of strategies, in addition to performance statistics on each individual strategy. For example, you might combine a trend-following moving average strategy with a short-selling strategy based on Aroon Oscillators, all in the same framework project.

Here is a screenshot of what a partitioned framework looks like in the Solution Explorer panel. Notice the presence of additional Money, Risk, and Exposure managers at the metastrategy level, so you can control the combined efforts of the two individual strategies shown in the screenshot. Typically, the strategy-level components take care of the strategy, and metastrategy-level components take a global view of all strategies so they can take care of the overall portfolio.



**Notice that you cannot export a strategy from a partitioned framework for use in a CATS system—there is no Export option on the right-click context menu for component strategies.** Only strategies from integrated framework projects can be exported for use in a CATS system.

## 3.5 Strategy Components and MetaStrategy Components

The IDE creates components at both the metastrategy level and at the strategy level. This means that in a 2-strategy metastrategy project such as the one shown above, you will have slots for 3 Money Manager components—one to manage trade Money for each of the 2 strategies at the strategy level, and a third one to manage money for the whole portfolio at the metastrategy level. See the screenshot above, and count the money manager components (3).

For example, each strategy money manager might permit half of your capital to be put into each technology instrument trade, whereas the metastrategy money manager might reduce the trade size of each strategy manager to a smaller value because there can be no more than 20% of the whole portfolio invested in technology companies.

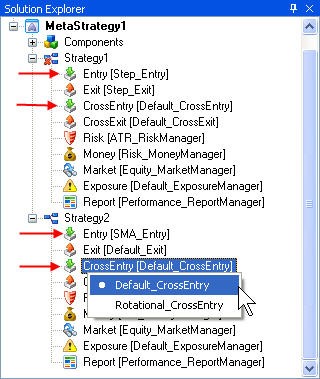
## 3.6 Normal and Cross Components

With both of the partitioned and integrated framework approaches, you can choose normal components (such as Entry) or cross components (such as CrossEntry, or ATSCrossComponent).

A normal component can only “see” (receive) bars for a single instrument. A cross component can see (receive) bars for multiple instruments. So if you intend to trade multiple instruments in a way that requires your code to use bars for both instruments simultaneously (such as for a correlation matrix for two instruments), then you should use cross components in your projects.

To be clear, you can trade multiple instruments using normal components—but each normal component will receive bars for only one instrument. Technically, at strategy start time the framework creates a separate instance of each normal component for each instrument that is traded by the strategy. In contrast, if you are using Cross components, only a single instance of each cross component is created for the whole strategy (one cross component, regardless of the number of instruments).

Here is a screenshot that shows Normal and Cross Entry components for a strategy that does not need to use multiple instruments at once (no Cross components required). In this example, the normal Entry components (Step Entry, SMA Entry) carry the useful code, and the cross components are empty (Default\_CrossEntry). This picture also shows how to right click the component type to choose a component from the drop down menu.



Technically, a new normal Entry component instance is created at strategy boot time for each instrument that is listed in the MarketManager component. This effectively limits the visibility of each component instance to the particular instrument that the component was created for. In contrast, only one CrossEntry or ATSCrossComponent is created for the whole strategy at boot time, so that single CrossEntry instance can “see’ (that is, will receive bars for) all instruments that are listed in the MarketManager component.

Programmatically, it is awkward to reference instrument BB from within a component that was instantiated to receive bars for only instrument AA. So if you want to reference bars for multiple instruments from within a single component, you should probably be using a cross component to make your programming life easier.

**BEWARE—by default, the IDE creates meta-strategy projects that do not work on the example data. This is because newly created projects use only Default\_XXX (that is, empty) components that do nothing.**

So to get a new strategy project to work, you must go through the component types and select working components. To begin with, try this set of components: Daily\_Simulation, Step\_Entry, Step\_Exit, Equity\_MarketManager, Flat\_MoneyManager, and EnableAll\_ReportManager. This combination will produce something useful to look at when you run the demo project strategies and view the performance statistics or bar charts after the test run (by right clicking the strategy node in the Solution Explorer panel and selecting something to view).

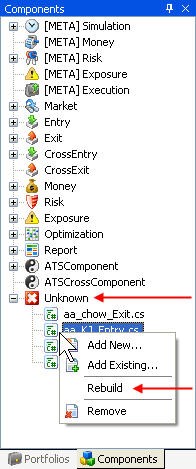
## 3.7 Unknown Components

The IDE contains a component category named “Unknown” at the bottom of its component types list in the Component panel. Perhaps a better name would be “Broken” or “Broken Components,” because that’s what this component category contains—components that will not compile.

For example, if you right click the Entry category to create a new entry component, and then make a coding error in the new component and close its editor window before you fix the error, your component will apparently disappear into thin air. It will no longer be listed in the Entry category where it was before, with all the other Entry components.

In operation, the IDE detects broken components at boot time, and moves them into the Unknown category without telling you. So to fix the situation, you must open the Unknown category, double click the broken component, edit it to fix the code, and then right click the component filename in the Unknown category to rebuild the fixed component. If the rebuild is successful, the IDE will move the working component back into its proper category. If not, try to fix the component again.

Here is a picture of the IDE Component panel, with the Unknown components expanded, and with a right-click context menu shown. You must use the context menu Rebuild option to rebuild broken components.



## 3.8 Component Templates

When you create a new component in the IDE, it creates a new skeleton component for you from a predefined code template for that type of component. The advantage of this approach is that the generated component skeleton contains some useful code in it, to save you typing. For example, you won’t have to type in namespace references, the class header, or the usual event handlers.

Only a subset of the possible event handlers are generated into new components, because you probably don’t need to use all of them (in which case you would end up having to do extra typing to delete unwanted event handlers). If the default set of generated event handlers are not sufficient for your strategy, you can easily type in the extra event handlers that you require.

# 4 SmartQuant Integrated Framework Structure

This section explains the code structure and general operation of a simple strategy project that uses an integrated framework. The section talks about a complete code example, typical strategy events, and how to place buy and sell orders from within an integrated framework. (Placing orders is one of the major differences between partitioned and integrated frameworks.)

We explain the integrated framework first because it is simpler to understand than the partitioned model. This is because integrated frameworks have fewer “moving parts” to explain than do partitioned framework examples. (You can check this yourself by counting the pages in this document devoted to each of the framework types.)

## 4.1 Section Outline

In what follows, we show a complete working ATSComponent for an integrated framework, so that you can see the big picture of what an integrated framework component looks like.

Next, we’ll talk about the code structure of the component. This means we’ll talk about the main sections themselves—declarations, initialization, event handlers, helper code—and what general kinds of code go in those sections, rather than explaining every line of code in the sections. We’re interested in structure here, not in the code details of this particular strategy.

Then we’ll talk about common events, and where and why you should use them. In the process of explaining these events, you will get a feel for how the code works to implement the strategy.

Finally, we’ll talk a bit about how to place orders in integrated frameworks by creating order objects and sending them directly to the execution provider (broker). Remember that order placement is very different in the two frameworks—partitioned frameworks use signals, and integrated frameworks use direct orders.

In a very real way, integrated frameworks are easier to program than partitioned frameworks because there are far fewer files to work with in your editor. Even for strategies of medium complexity, almost everything (such as all the event handlers) can be in a single convenient file.

Now let’s begin by looking a complete code example for an integrated framework.

## 4.2 Using Daily Bars for Writing Convenience

For easy understanding, many examples in this document are presented using daily bars, because it makes intuitive sense that OnBarOpen could correspond to the idea of “on market open at the beginning of the day.” This correspondence is easier for novice traders to understand.

Using daily bars also makes it easier to say things like “yesterday’s close” and “today’s high” instead of “the close of the previous bar” or “the high value of the current bar.” So fewer words and shorter sentences are required if daily bars can be used to help express the ideas.

But using daily bars for writing convenience does not limit the SmartQuant framework in any way. Be assured, the SmartQuant system is a real time system that is intended for trading bars of any size, all the way from fast tick data to slower weekly bars, and beyond.

## 4.3 A Complete Code Example (Integrated Framework)

Here is a simple code example for an integrated framework.

The following code example implements a very simple gap closing strategy. It opens a new position when a downward gap in prices is formed, and closes the position when the gap closes or at the end of the trading day, whichever comes first.

Every morning at market open (in OnBarOpen), this strategy looks for a downward gap size of 2% between yesterday’s close and today’s open. When the strategy spots such a gap, it buys 100 shares long to open a new position. If the new position is successfully opened, the strategy automatically enters a limit order (in OnPositionOpened) to sell if the stock price recovers to the level of yesterday’s close during the trading day (thereby closing the gap). If today’s bar arrives (in OnBar) and the open position still exists (the gap was not closed), the limit order is cancelled, and a market order is issued to close the position.

Like many C# code files, the following SmartQuant integrated framework code file is organized into four major sections. (We identify these four parts with big comments in this example to emphasize the code structure for novice programmers.) The four sections are:

* Assembly references (“using…”),
* Class / variable / parameter declarations, and
* Event handlers (OnBarOpen, OnBar, OnPositionOpened, etc)
* Helper classes and methods (none in this example)

//

// Part 1 NAMESPACE REFERENCES

//

// Windows namespace references using System; using System.Drawing; using System.ComponentModel;

// SmartQuant namespace references using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

//

// Part 2 CLASS HEADERS, VARIABLE DECLARATIONS, INITIALIZATION

//

// attribute header and class header

[StrategyComponent ("{9e473d29-3638-402c-8a9e-5c447608d165}",

ComponentType.ATSComponent,

Name = "Tech1\_System1\_ATS",Description = "")]

public class

**Tech1\_System1\_ATS : ATSComponent {**

// class variable declarations

private double prevClose; // updated in OnBar private double desired\_gap = 0.02; // gap size of 2 percent private double qty = 100; // n shares to order private SingleOrder sellOrder;

// parameter declarations (these show up in the IDE properties panel)

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

// dummy init routine to show you where it goes in the file

// (it is not required for this example strategy) public override void

**Init () {**

// how to draw a simple moving average on the price chart

// sma = new SMA (Bar, smaLength);

// sma.Color = Color.Blue; // Draw (sma, 0);

}

//

// Part 3 EVENT HANDLERS (most code goes here for simple strategies)

//

// event handlers for buying and selling public override void **OnBarOpen (Bar bar) {**

// calculate the size of the gap down since last bar double gap\_size = (prevClose - bar.Open) / prevClose; // issue a buy order if the gap is big enough if (gap\_size > desired\_size) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech1\_System1 - Buy"; order.Send ();

}

}

public override void **OnPositionOpened () {**

// issue a stop limit order whenever a position is opened sellOrder = LimitOrder (Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System1 - Sell"; sellOrder.Send ();

}

public override void **OnBar (Bar bar) {**

// update the prevClose to be today’s close prevClose = bar.Close;

// if we still have a position open, close the position if (HasPosition) { sellOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech1\_System1 - Sell"; order.Send ();

}

}

//

// Part 4 HELPER CODE

//

// but this strategy is so simple it doesn’t need any helper functions

}

## 4.4 Defining Variables, Properties, and Parameters

As usual in normal C# code files, you define code variables at the top of integrated framework (ATSComponent) files, as shown above and below.

You can also treat variable values as user-defined property values that users can set through the Properties panel in the IDE or CATS systems. To do this, first make your variable into a true property with Get/Set interfaces. Then add the C# Category and Description attributes to the property using square brackets […] as shown below.

Now when the strategy is loaded into a SmartQuant program, users can set initial variable values for your strategy by selecting the component and then using the Properties panel to set their values of choice. Once you set a property value in the Properties panel, the IDE will save the specified value when you close the solution, so that when you reload the solution next time, your specified value will still be there.

// class variable declarations

private double prevClose; // previous bar close private double percent = 2; // gap size in percent private double qty = 100; // n contracts to order private SingleOrder sellOrder;

// parameter declarations (these show up in the IDE properties panel)

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

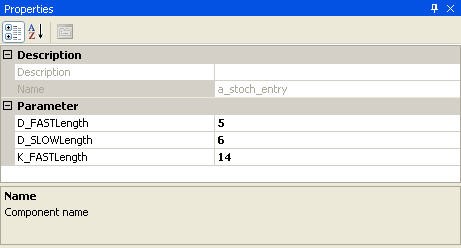
When you define your variables as properties as shown above, they become visible in the IDE and CATS systems as user-settable properties in the Properties panel. Here is some example code, and a screenshot that shows how the description attributes are displayed in the Properties panel. First, here is some code (code for one property is shown, but 3 appear in the Properties panel).

[Category ("Parameter"), Description ("Length of K\_FAST")]

[OptimizationParameter (5, 25, 1)] public int K\_FASTLength { get { return k\_fastLength; } set { k\_fastLength = value; }

}

Next, here is a screenshot of the corresponding Properties panel that shows (1) the Category header “Parameter”, and (2) three properties named D\_FASTLength, D\_SLOWLength, and K\_FASTLength. Notice that the third line in the Parameters Category below corresponds to the code immediately above, for the K\_FastLength property. You can choose your own Category names other than “Parameter”, of course, to suit the natural classifications of your strategy.



## 4.5 The Init Method

The Init method is where your code creates new objects and attaches them to class variable declarations, and where you specify what series to draw on price charts. The Init method is called by the SmartQuant framework before the strategy itself is started.

Here is example Init code that creates a new SMA (simple moving average) object for the default bar series, colors the moving average line Blue, and then draws the moving average line on the default price chart (pad 0). If you want a line on the volume chart, use pad 1. If you want it on a chart of its own, use pad 2 (or some other number greater than 1).

private SMA sma; // simple moving average variable private smaLength; // length of moving average

public override void

**Init () {**

// how to draw a simple moving average on the price chart sma = new SMA (Bar, smaLength); sma.Color = Color.Blue;

Draw (sma, 0);

}

The SMA object constructor takes two parameters—a reference to the BarSeries (a time series of prices) that should be averaged, and the length of the moving average to calculate. Here, we use the name “Bar” to refer to the default bar series (object type BarSeries) for this ATS component.

“Bar” is actually a property of the ATSComponent base class; it returns the current instrument series that the component is working with, so you don’t have to know the name of the instrument in order to request its bar series in your code. In some strategies, you might want to create your own BarSeries objects, in which case you would pass your own BarSeries to SMA constructors.

If you want to draw averages (or other technical indicator lines) on pads below the price chart, just increment the pad number to something greater than zero. Pad 0 is the price chart. Pad 1 is the volume chart. Pads 2+ can be whatever you want them to be.

The code above can be used in either partitioned or integrated frameworks.

## 4.6 Initialization

This section shows you how and where to declare variables and strategy parameters (as usersettable properties in the Properties panel of the IDE or CATS applications).

#### 4.6.1 Defining Variables and Parameters

As usual in C# code files, you can define arbitrary code variables at the top of either partitioned or integrated framework files, as shown below.

You can also treat variables as properties that users can set through the Properties panel in the IDE or CATS systems. To do this, first make your variable into a true property with Get/Set functions. Then add the C# Category and Description attributes to the property using square brackets […] as shown in the examples below.

Now when the strategy is loaded into a SmartQuant program, users can set initial variable values for your strategy by right clicking the component and using the Properties panel.

// class variable declarations

private double prevClose; // previous bar close private double percent = 2; // gap size in percent private double qty = 100; // n contracts to order private SingleOrder sellOrder;

// parameter declarations (these show up in the IDE properties panel)

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

#### 4.6.2 Optimizing Parameters

The IDE can optimize the values of variables that have been coded as properties as shown above. Optimization is useful for determining the approximate range of values that generate the most profit for your strategy. But you should be careful not to depend too much on “over fitting” parameters to any one set of market data—this practice can lead to brittle trading systems that fail on any other market data that is not like the data used for optimization.

One approach to optimization is to optimize your parameters, and then manually change them to some nearby “rounded” number that is not exactly the one picked by the optimizer. At least this way your strategy will not get to depend on optimized parameter values to three decimal places.

To code a parameter for optimization, you must add the Optimize attribute to the attributes of the property declaration beside the Category and Description attributes, as shown below. You must also specify a range of values to use for optimization—one value for the start of the range, one value for the end of the range, and a third value for increments within the range.

Here is a code fragment that shows how to set up the length of an exponential moving average for optimization. The default length of the average is 3, set when the variable is declared. The optimization range begins at 3, ends at 20, and increments by 1. This means the optimizer will run your strategy 17 times (3 to 20 by 1) during optimization, and then choose the best value for you.

// the default length of a short exponential moving average private int length\_short = 3;

// optimization range is 3,4,5,..,20

[Category ("Parameter"), Description ("Length of Short EMA")]

[OptimizationParameter (3, 20, 1)] public int Length\_short { get { return length\_short; } set { length\_short = value; }

}

## 4.7 ATS Component Properties, Methods, and Events

Here is a list of interfaces provided by the ATSComponent object. This list will help you to see how powerful this strategy object is, and give you a flavor for the kinds of things that are modeled by the SmartQuant system.

More detailed listings and descriptions for the (many dozens) of classes in the SmartQuant system are provided in the API Documentation.

As a practical matter, you will probably put most of your essential strategy code in the various event handlers below. (Of course, you are likely to have many utility and supporting methods too, depending on the complexity of your strategies.) So it is useful to have a long list of event handlers that you can use, as shown by the list below.

**Public Properties**

// Name Description

Bar Returns default bar series for component

Bars Returns a list of bar series that can be

Indexed by instrument, bar type, and bar size

Description (inherited from ComponentBase)

Global (inherited from StrategyBaseSingleComponent) HasPosition Returns true if a positions is open for

the current instrument

Instrument Returns the current instrument Name (inherited from ComponentBase)

Orders Returns outstanding orders for strategy Portfolio Returns portfolio for strategy

Position Returns position for current instrument

Strategy Returns parent strategy of this component

**Public Methods**

// Name Description

Connect To a provider

Disconnect From a provider

Draw A series on the main IDE window

Equals (inherited from Object)

Determines whether the specified Object is equal to the current Object.

GetHashCode (inherited from Object) A hash function for a particular type.

Can be used by hashing algorithms.

GetType (inherited from Object)

Gets the Type of the current instance.

Init for initializing your strategy

LimitOrder Creates a limit order

MarketOrder Creates a market order

**Event Handlers**

// Name Description

OnBar Called on trailing edge of every bar

OnBarOpen Called on leading edge of every bar

OnBarSlice Called after all bars for all instruments in a particular time slice have been emitted

OnCorporateAction Called when a corporate action event arrives

|  |  |
| --- | --- |
| OnFundamental | Called when a fundamental data event arrives |
| OnMarketData | Called when a market data event arrives |
| OnMarketDepth | Called on new market depth (Level II) quotes |
| OnNewOrder | Called on orders sent to execution provider |
| OnNews | Called when a news event is received |

OnOrderCancelled Called when an order is cancelled

OnOrderDone Called when an order completes

OnOrderFilled Called when an order is completely filled

OnOrderRejected Called when broker rejects an order

OnOrderStatusChanged Called when an order status is updated

OnPositionChanged Called when position data changes

OnPositionClosed Called when a position is closed

OnPositionOpened Called when a position is opened OnPositionValueChanged Called when market value of current

position changes

OnQuote Called on trailing edge of every quote

OnStopCanceled Called whenever an internal stop is cancelled

OnStopExecuted Called whenever an internal stop is executed

OnStopStatusChanged Called whenever a stop changes its status

OnStrategyStop Called when the strategy is stopped OnTrade Called on trailing edge of every trade received

**Order Types and Operations**

// Name Description

Sell Creates a sell order

SellLimit Creates a sell limit order

SellStop Creates a sell stop order

SellStopLimit Creates a sell stop limit order

SendLimitOrder Sends limit order to broker for execution

SendMarketOrder Sends market order to broker for execution

SendStopLimitOrder Sends stop limit order to broker for execution

SendStopOrder Sends stop order to broker for execution

SetStop Sets internal stop (trailing or absolute)

StopLimitOrder Creates a stop limit order

StopOrder Creates a stop order

TrailingStopOrder Creates a trailing stop order.

ToString Returns a printable string for current object.

## 4.8 Common Strategy Events

Here is a very short list of useful event handlers that are called during a typical trading session, for both partitioned and integrated frameworks. This list is very minimal, and is intended only to give you a simple understanding of the main event handlers (and their typical uses) in a strategy. Of course you might end up using many more kinds of event handlers in other ways, depending on the complexity of your strategy.

* OnBarOpen – called on leading edge of all bars (e.g. for buy at market open, daily bars)
* OnBar – called on trailing edge of all bars (e.g. for buy at market close on daily bars)
* OnBarSlice – called when all per-instrument bars have been emitted
* OnPositionOpened – called whenever a new position is confirmed opened
* OnPositionChanged – called whenever a position is increased or decreased
* OnValueChanged – called when portfolio value changes (which is with every quote!)

The main idea of the SmartQuant event model is to match how things actually happen in the real trading world. Events do a good job of modeling changing prices, changing orders, changing order statuses, changing positions, and changing position values.

Several kinds of events are fired in the SmartQuant framework during the course of buy-sell transactions involving market data providers, execution providers, and your strategies. Some (pre-trade) events are fired to model market data provider actions, some (trade) events are fired to model execution provider actions, and still other (post-trade) events are fired to model updates to portfolios and positions.

A strategy can subscribe and listen to all of these kinds of events, so that a strategy event handler (named OnXXX, where XXX is the event name) can be called to implement a strategy action when the event fires. For example, when an event for a trade in MSFT is fired, the SmartQuant framework invokes the appropriate event handler in the appropriate component (that was created for the MSFT instrument).

Because the framework automatically invokes the event handler in the right component for the instrument, your strategy code can be much simpler—you don’t have to worry about indexing all actions with if-else clauses that select the right code for the current instrument. Instead, the framework does that for you. You can usually write most of your code without referencing any particular instruments, because a separate instance of each component is created for each instrument that is traded by the strategy.

#### 4.8.1 OnBarOpen

OnBarOpen is fired before every bar that a BarFactory produces. (Each market data provider has a BarFactory that can be configured in the Properties panel.)

The purpose of OnBarOpen is to give you a chance to do something on the leading edge of a bar, before the bar has been created. (In contrast, OnBar is fired on the trailing edge of a bar.)

For example, assuming that daily bars are being used, OnBarOpen is the place where you would put code that wants to buy or sell when the market first opens. This is because for daily bars, you only get one bar for the whole trading day. So you want your “buy at the open” action to take place on the leading edge of the day’s bar (using OnBarOpen), rather than on the trailing edge of the day’s bar (using OnBar).

If daily bars are not being used, then OnBarOpen just becomes a place where you can take action on the leading edge (the “open”) of the next bar that will be produced.

In many cases where buying at the open is not important, your code won’t need to use this event.

#### 4.8.2 OnPositionOpened

OnPositionOpened is fired whenever a new position is established as a result of a completed trade. The trade order must complete in order for a position to be opened. All positions must have non-zero position sizes in them. When a position size is reduced to zero (by closing the position with appropriate trade orders), the position objects are destroyed. All that remains are logs of all trades that established (or tried to establish) the position, and trades that closed the position, if it was ever created.

Here is an example of how to automatically create a stop limit order when a long position is first created. The code below is from an integrated framework, so it constructs a true sell order object and sends it directly to an execution provider for trade execution.

“Qty” means “quantity”, and is a positive integer that specifies the size of the trade (number of shares, or contracts). This code creates a stop loss order using the same Qty that was used in the opening buy order, and uses the previous bar closing price as the limit price.

// create and issue a sell limit order in an integrated framework public override void **OnPositionOpened () {**

sellOrder = LimitOrder (Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System1 - Sell"; sellOrder.Send ();

}

The code above should not be used in partitioned frameworks; you should use signals instead.

#### 4.8.3 OnValueChanged

OnValueChanged is fired whenever a new incoming trade price changes the value of a position. The price is taken from an actual trade (the “last price” on most trading consoles), not from the current bid or ask prices. Accordingly, this event will fire, and the value of your affected positions will change, many more times than OnBar events occur. This is because quote data is unaggregated data, whereas bars are aggregated data—one bar is produced for many quotes.

OnValueChanged is a good place to adjust positions or trading orders, if they depend on the overall value of the portfolio components, rather than just on the price of the current bar. For example, if you wanted to issue trade orders to close all positions if the value of the portfolio falls below $10,000.00, this would be a good place to put that code.

#### 4.8.4 OnPositionChanged

OnPositionChanged is fired whenever the size of a position is changed, one way or the other. For example, this event will fire every time a partial fill confirmation updates the position size.

OnPositionChanged is a good place to adjust the size of stop loss orders that must track partial fills. Every time a new partial fill confirmation comes in, you can add the confirmed quantity to the size of your outstanding stop loss order. This way, your stop loss order will more accurately reflect the actual fill status of your original order.

#### 4.8.5 OnBar

OnBar is fired whenever the market data provider BarFactory completes the calculation of a new bar (which is aggregated from raw quote and trade data). Each bar contains OHLCV (open high low close volume) data for the past bar interval. Accordingly, the OnBar event fires on the trailing edge of the current bar. If you need to take action on the leading edge of the next bar, use the OnBarOpen event.

OnBar is a good place to put your routine strategy code that works with each new bar in a series of bars, such as for comparing the current bar prices with past bar prices, for calculating or updating technical indicators with new bar data, for calculating moving averages, for making trading decisions, and for issuing trading signals or orders.

#### 4.8.6 OnBarSlice

OnBarSlice fires whenever the market data provider BarFactory has finished emitting new bars for all the instruments in the current portfolio. BarFactory emits new bars sequentially when multiple instruments are being traded—first a bar for instrument 1, then a bar for instrument 2, and so on. After all individual bars have been emitted, BarFactory will fire OnBarSlice to tell strategy components that the end of the instrument list has been reached, and that all new bars for the current bar interval have been emitted.

OnBarSlice solves the problem of strategy code not knowing when all the instrument bars for a correlation strategy are present and ready to be used. For example, if you try to do correlation calculations in the OnBar event handler for multiple instruments, your code will have to keep track of which instruments have been seen in the OnBar handler for the current bar time interval. First your code would mark down (or save the bar data from) the first bar for the first instrument in the correlation, ask itself if all bars are present, and exit if not. Finally, when the bar for the last instrument in the correlation arrives, your code can get on with doing correlation calculations.

In contrast, using OnBarSlice is much easier. When execution arrives in the OnBarSlice event handler, your code can just start doing correlation calculations immediately—no need to mess with keeping track of instruments and bars in the correlation set. OnBarSlice guarantees that all bars for all relevant instruments have been emitted.

## 4.9 Signals in the Integrated Framework

Integrated frameworks do not use trading signals. Only partitioned frameworks use signals.

Instead, code in an integrated framework creates orders and sends them directly to the execution provider for execution. This means that in an integrated framework, you have more flexibility to create and manage trading orders, but your code also has to do more work.

## 4.10 How to Place Orders (Integrated Framework)

Here are short code fragments that show how to place typical trading orders to buy/sell long, and sell/cover short for the integrated strategy framework. These fragments are just a combination of code examples that show the general syntax of orders. Sometimes multiple statements with slightly different syntaxes are used, just to show more than one possible way of doing things.

Here is a list of order types and actions for integrated frameworks: Buy, BuyLimit, BuyStop, BuyStopLimit, Sell, SellLimit, SellStop, SellStopLimit, SendLimitOrder, SendMarketOrder, SendStopLimitOrder, SendStopOrder, StopLimitOrder, StopOrder, and TrailingStopOrder. Most of these orders have similar syntax (see the API reference for more details).

Here is some simple example code that shows how to buy and sell long, using orders sent directly to the execution provider. (To sell and cover short, reverse the sequence of the buy/sell orders).

// MarketOrder var type holds only market order objects MarketOrder myOrder;

myOrder = MarketOrder (Side.Buy, qty);

// another example, this time on the sell side myOrder = MarketOrder (Side.Sell, qty); myOrder.Text = "this text shows in the trade log"; myOrder.Send ();

// SingleOrder var type holds any order type object (long,shrt,mkt,limit) SingleOrder myOrder;

myOrder = LimitOrder (Side.Sell, qty, prevClose);

// another syntax, showing more arguments

myOrder = LimitOrder (position.Instrument, Side.Sell, qty, prevClose); myOrder.Text = "Tech1\_System1 - Sell"; myOrder.Send ();

// how to cancel an order that has been sent previously myOrder.Cancel ();

Notice that a *MarketOrder* variable type is used to hold a market order object, whereas a *SingleOrder* variable type is used to hold a LimitOrder in the second example. *SingleOrder* is a generic variable type that can be used to hold market, limit, stop, and stop limit order objects.

You can also use variable types *MarketOrder, LimitOrder, StopOrder, or StopLimitOrder* if you like, to hold objects of the corresponding type, as shown below:

myOrder = MarketOrder (…) myOrder = LimitOrder (…) myOrder = StopOrder (…) myOrder = StopLimitOrder (…)

The code above is for integrated frameworks—you should use signals in partitioned frameworks!

Notice that the code examples above do not call the “new” operator to create a new order object. Instead, the code examples call methods of the ATSComponent class. These methods (such as MarketOrder) create an order, insert default values for the current strategy Execution Provider and strategy portfolio, and then register the order with the strategy.

# 5 SmartQuant Partitioned Framework Structure

This section introduces some typical code blocks for partitioned framework projects. Once you understand these basic building blocks, we can move on to explaining how to fit them all together to create working strategies for various trading theories.

The following examples show how to code the following actions in partitioned frameworks:

* how to code which instruments to trade (Market Manager)
* how to code an entry to a new position (Entry)
* how to code an exit from a position (Exit)
* how to code the size of a trade (Money Manager)
* how to help to calculate trade size based on risk factors (Risk Manager)
* how to control overall portfolio exposure (Exposure Manager)

## 5.1 Components in the Partitioned Framework

This section explains some of the high level goals and structure of components in the partitioned framework model. Keep in mind that components (files) are primarily packaging mechanisms that simply group related bits of strategy code into one file. However, by design it also happens that there is a specific component defined for each major decision making step in a strategy, so that logic for each major decision making step can be placed into an interchangeable component.

#### 5.1.1 The Main Goal of Components

The most important goal of components in the partitioned framework is to represent a complete and interchangeable block of code logic in the strategy decision making workflow. This means that components act as packaging containers for related kinds of trading logic.

For example, the Entry component should contain all code relating to entering a new position, the Exit component should contain all code relating to exiting a position, and the Money component should contain all logic for controlling the size of a trade in a particular instrument. If you maintain these logic boundaries in your code, then you will be able to easily mix and match various components (that implement different Entry, Exit, Money, and Exposure strategies) to explore various combinations of strategies.

Notice that you cannot easily mix and match or swap strategy logic in the integrated framework, because then you would have to cut and paste code, or write code logic to enable/disable chunks of code that implemented different pieces of strategy logic (ugh). So if you want to mix and match and explore combinations of strategy logic blocks, the partitioned framework can help you.

#### 5.1.2 Control Flow among Components

When you create a new partitioned framework (MetaStrategy) project and add a strategy project to it, the IDE automatically creates a set of default components for you. But how do trading signals and control flows pass among the components? How do you know where to put your strategy code, and how to get it executed?

The general answer to these questions is that (1) components are packaging devices only, and (2) control and signal flow among components depends on the various method and event names that are available within each component type.

This means that in order to get your code executed as part of the normal signal flow, you must implement one or more methods or event handlers *that use predefined method names or event names that are part of the partitioned framework model*.

In strategy level components, you will probably use event handlers for most of your strategy logic. The events are mostly straightforward—OnBarOpen, OnBar, OnPositionOpened, etc. Many examples of these common events are given in this document, and their runtime operation is easy to understand, so you probably won’t have any problem understanding how events work. As usual, your event handler will be called when the associated event is fired.

But control and signal flow among metastrategy and strategy components is somewhat more complex, because not all control flow is implemented using events. Instead, some important control flow is implemented using specific method names that you must provide or implement if you want to take some actions at the metastrategy level.

Often when you are testing out combinations of strategy level ideas, you probably won’t bother implementing much logic at the metastrategy level, so you usually won’t have to worry about how control flow actually proceeds from strategy to metastrategy level and back again. Instead, it is probable that you will save your serious metastrategy logic for the integrated framework model on ATSMetaStrategy projects. For this reason, the details of metastrategy control flow are not explained here. The details are somewhat complex, and are not worth explaining here in this introductory document.

There is a section below on metastrategy components that provides more detailed information on their design goals, their special method names, and how their special methods should work.

## 5.2 Signals in the Partitioned Framework

Partitioned frameworks use “signals” to initiate trades, instead of sending direct orders to an execution provider (broker). It might help to think of signals as “suggestions for trades” that various strategy components can modify or reject before the signals actually become true trading orders in the Execution component of a partitioned framework.

The main advantage of using signals is that they make it easier to initiate and manage simple trading actions among separate components. The partitioned framework can do more of the order management functions for you, and your code is simpler. For example, the partitioned framework automatically fills in signal parameters such as the target execution provider name (the broker) and the portfolio to use to store the trade results. (These two values come from strategy level properties Strategy.ExecutionProvider and Strategy.Portfolio.)

#### 5.2.1 Signal Types

Here is a list of signals that you can send from Entry and Exit components.

* LongEntry – open a new long position in current instrument
* LongExit – close an existing position in current instrument
* ShortEntry – open a new short position in current instrument
* ShortExit – close an existing position in current instrument
* Buy – buy the current instrument
* BuyLimit – buy the current instrument with a limit on price
* BuyMarket – buy the current instrument at market price
* BuyStop – buy the current instrument at market when the stop price is reached
* BuyStopLimit – buy the current instrument with limit when stop price is reached
* Sell – sell the current instrument
* SellLimit – sell the current instrument with a limit on price
* SellMarket – sell the current instrument at market price
* SellStop – sell the current instrument at market when the stop price is reached
* SellStopLimit – sell the current instrument with limit when stop price is reached

Each signal type accepts combinations of different parameters, such as instrument to trade, time the order is in force (e.g. good until cancelled), stop and limit prices, and text strings to help identify and describe orders in log files.

There are checks in the framework code for exits (LongExit, ShortExit) that will reject the exit if you don’t have an existing position to close. But your strategy should still be smart enough not to call an exit method unless you have an open position to close.

#### 5.2.2 Signal Flow among Components

The partitioned framework automatically routes trading signals (suggestions) through various components in the signal chain. For example, when you initiate a buy action with “LongEntry()” code in an Entry component, and the partitioned framework allocates a signal object and routes it through all the components that should see the signal before the Execution component converts the signal to a trading order and sends it to a broker.

Here is an ordered list of components that will have a chance to modify or reject a trading signal that originates in an Entry or Exit component:

* Signal originates in strategy code (“LongEntry()” or “LongExit()”)
* Then to strategy level Money Manager (can call strategy Risk Manager helper)
* Then to strategy level Exposure Manager (either totally accepts or rejects the signal)
* Then to metastrategy level Money Manager (can call metastrategy Risk Manager helper)
* Then to metastrategy level Exposure Manager (either totally accepts or rejects the signal)
* Then to metastrategy level Execution Manager (signal converted to order, sent to broker)

Methods in risk manager components are never called automatically by the framework as part of the normal signal flow. Instead, risk manager methods must be explicitly called by one of the other Money or Exposure components. Event handlers are different—any event handlers that you write in risk manager components are automatically called when the associated events fire.

## 5.3 Market Manager Component

A MarketManager component (in a partitioned framework) determines the set of instruments that is seen and processed by a strategy. For example, a market manger component might specify that data for the “MSFT” instrument be provided to the current strategy.

The market data provider can be either a simulated provider that reads from a local SmartQuant database, or it can be a real market data provider that provides real time market data. In either case, the data provider contains a BarFactory that can manufacture bars from the incoming data that it has access to (either a series in a database, or real time tick or trade data).

Here are two code fragments for MarketManager components that show typical ways of specifying the instruments that you want to use in your strategy. In the first example you simply name the symbols explicitly, and in the other you specify desired instruments by their object reference in the InstrumentManager. In both cases, you tell your strategy to “AddInstrument” to add the instrument to the list of instruments that the strategy will process at runtime.

// specify the instruments to use in this strategy public override void

**Init() {**

AddInstrument("MSFT");

AddInstrument("CSCO");

}

// use all equity instruments in InstrumentManager for this strategy public override void

**Init() {**

foreach(Instrument inst in InstrumentManager.Instruments) {

if (inst.SecurityType == SecurityType.CommonStock)

AddInstrument(inst);

}

}

The Init method above is the usually the only method that appears in a MarketManager component. The rest of the code for a MarketManager class is created for you by the IDE (that is, the namespace references, the class header, and other code in the template for the component.

## 5.4 Entry Component

The Entry component is responsible for determining when to enter new trades. Typically you enter a trade using an event handler named OnBarOpen or OnBar. The OnBarOpen event fires on the leading edge of every bar, and the OnBar event fires on the trailing edge of each bar. For example, if daily bars were being used, OnBarOpen would fire at the market open (when the first trade was made at the exchange), and OnBar would fire at the closing bell (end of trading day).

Here is an example that shows how to enter a new trade on a price breakout that is greater than the highest high of the past 4 weeks (20 daily bars). Of course, there is nothing in this example that says daily bars are being used—the example will work with bars of any size. I only used daily bars to make the explanation easier. (Daily bars are intuitively understood by most people.)

// vars for breakout period length, and highest/lowest values private int period = 20; private double high; private double low;

// enable trading actions only after “period” days have gone by private bool enabled = false;

public override void **OnBar (Bar bar) {**

// go long or short on a breakout, depending on breakout direction if (enabled) { if (bar.High > high) LongEntry (); if (bar.Low < low) ShortEntry ();

}

// update highest/lowest values only after “period” bars are seen if (Bar.Count >= period) { high = Bar.HighestHigh (period); low = Bar.LowestLow (period);

// enable trading actions after high/low values are set enabled = true;

}

}

public override void **OnTrade (Trade trade) {**

// go long or short on a breakout, depending on breakout direction if (enabled) {

if (trade.Price > high)

LongEntry (); if (trade.Price < low)

ShortEntry ();

}

}

Notice that this entry component uses both OnBar and OnTrade to watch for breakouts. OnTrade is only required if your strategy is listening to trade data from your market provider. Remember that trade data is *unaggregated* data, whereas bars are *aggregated* data that are manufactured by someone—either they are manufactured by your market data provider before reaching your computer, or they are manufactured by the BarFactory of your market data provider. (The

BarFactory can be configured using the Properties panel of the IDE or CATS system.)

#### 5.4.1 How to Place Orders using Signals

Here are code fragments that show how to place typical trading orders to buy/sell long, and sell/cover short. These examples are for the partitioned framework, and are considerably different than those for the integrated framework.

These code examples are back to back examples of various syntaxes for various kinds of orders, just to give you a feel for what orders look like in the partitioned model.

The simplest way to buy and sell long at the market:

LongEntry ();

LongEntry (“this text shows in the trade log”);

LongExit();

LongExit(“this text shows in the trade log”);

The simplest way to sell and cover short, at the market:

ShortEntry ();

ShortEntry (“this text shows in the trade log”);

ShortExit();

ShortExit (“this text shows in the trade log”);

The code above can only be used in partitioned frameworks, because it uses signals. A complete list of orders that use signals was listed above in Signal Types.

## 5.5 Exit Component

The Exit component is responsible for determining when to exit existing trades. Often a strategy will use multiple exists, such as one for the underlying theory of the trade (such as a trend crossover), one for reaching a profit target, and one for a stop loss limit.

Here is an exit component for the breakout strategy just explained above in the Entry component section. The exit component is almost a copy of the entry component, with the exception of two notable things—(1) the exit period “look back” is shorter (10 days versus 20 days), and (2) the trading orders are different (LongExit versus LongEntry, etc).

It might take you a moment to puzzle out the exit logic. Keep in mind that a breakout exit strategy acts kind of like a trailing stop loss limit. If prices break out on the upside, a trade is entered long, and the lowest low in the exit look back period will start moving up as the prices move up, just like a trailing stop loss limit order does. Eventually, the current bar low will be less than the lowest low in the look back period, and so a LongExit signal will be given.

// vars for breakout period length, and highest/lowest values

// notice breakout exit period is SHORTER than breakout entry period private int period = 10; private double high; private double low;

// enable trading actions only after “period” days have gone by private bool enabled = false;

public override void **OnBar (Bar bar) {**  if (enabled) { if (bar.Low < low)

|  |  |  |
| --- | --- | --- |
| LongExit (); if (bar.High > high) |  | // LongExit |
| ShortExit (); |  | // ShortExit |

}

// update highest/lowest values only after “period” bars are seen if (Bar.Count >= period) { high = Bar.HighestHigh (period); low = Bar.LowestLow (period);

// enable trading actions after high/low values are set enabled = true;

}

}

public override void **OnTrade (Trade trade) {**  if (enabled) { if (trade.Price < low)

|  |  |  |
| --- | --- | --- |
| LongExit (); if (trade.Price > high) |  | // LongExit |
| ShortExit (); |  | // ShortExit |

}

}

## 5.6 Money Manager Component

A MoneyManager component (in a partitioned framework) determines a size for a trade—that is, the quantity the trade should buy or sell. Money manager components look at single trades (actually they look at single Signal objects), and often call a risk manager “helper” component for help with risk calculations. So while the money manager actually inserts the value for a trade size into the signal, a risk manager helper method will often be the code that specifies the exact size of the number to insert.

When a trading signal (LongEntry) is issued in an Entry or Exit component, the framework calls the GetPositionSize method in the Money component of your strategy. The name of this method is fixed, so all you need to do is add code for sizing the trade into the method body.

#### 5.6.1 Without Calling Risk Manager Helper Code

Here is a money manager code example that sets the size of new positions to a default size of 100, and sets the size of trades that close existing positions to the existing position size. This example does not call a risk manager helper component.

Notice the use of signal types: SignalSide.Buy, Sell, SellShort, and BuyCover. These signal types are not available in the integrated framework, because integrated frameworks do not use signals.

// default trade size is 100 shares, contracts, or whatever double default\_size = 100;

public override double

**GetPositionSize (Signal signal) {**

double positionSize = 0; switch (signal.Side) {

// trade size for new positions is default position size case SignalSide.Buy: case SignalSide.SellShort:

if (!HasPosition) positionSize = default\_size; break;

// trade size for closing positions is the existing position size case SignalSide.Sell: case SignalSide.BuyCover:

if (HasPosition) positionSize = Position.Qty; break; }

return (int)(positionSize);

}

#### 5.6.2 With Calling Risk Manager Helper Code

Next, here is a example that calls a risk manager helper method (GetPositionRisk) to calculate a position risk number that can be used by the money manager to determine a size for the trade. The money manager code does not know or care about how the risk calculation is carried out—all the money manager wants is a return value to help set the size of the current trade. See the next section for more information on how the helper method calculates a position risk number.

// risk 10% of portfolio on any particular position private double portfolioRisk = 0.1;

public override double **GetPositionSize (Signal signal) {**  switch (signal.Side) {

// new positions buy less if volatility (ATR) is high (greater than 1), // and buy more than the usual maximum if ATR is < 1. case SignalSide.Buy: case SignalSide.SellShort: if (!HasPosition) {

double shares; // initial allocation

double position\_fraction; // decreases as volatility increases

// the most you could buy without margin

shares = Portfolio.GetTotalEquity () / Instrument.Price ();

// use risk helper method GetPositionRisk (see next section) // denominator returned adjusts fraction above or below 1.0 position\_fraction = portfolioRisk / GetPositionRisk (); return (int)(shares \* position\_fraction);

}

break;

// close positions using existing position sizes case SignalSide.Sell: case SignalSide.BuyCover:

if (HasPosition)

return Position.Qty; break;

}

// if logic above doesn’t match, then set trade size to zero return 0;

}

}

## 5.7 Risk Manager Component

Risk manager components are helper components for the MoneyManager and ExposureManager components. They package up all the risk management code in one place, and look at things such as the existing market conditions, trade conditions, portfolio positions, or other data sources, and perform various risk calculations for the money and exposure managers.

#### 5.7.1 Default Risk Manager Component

When you create a new risk manager component at the strategy level, it only contains one empty event handler named OnPositionOpened. This handler is provided so that the strategy risk manager can automatically issue a stop loss order whenever a new position is opened.

Here is what the code for the default strategy level risk manager component looks like. You can manually add any other event handlers that you need to this component (such as OnBar, etc), if your strategy requires them.

public override void  **Init() {**

// Add your code here

}

public override void  **OnPositionOpened() {**

// Add your code here

}

Risk manager components can also accept / reject signals if you implement a Validate method. But you will have to implement this method yourself—it is not automatically generated for you.

public override bool  **Validate(Signal signal) {**

// return 0 to accept the signal, and 1 to reject it Console.WriteLine("Strategy Risk Manager"); return base.Validate(signal);

}

#### 5.7.2 A GetPositionRisk Example

The code example below is from a risk manager component that calculates a suggested position risk for a pending trade in the MoneyManager component earlier. It returns a calculated value to the money manager, which will use the value to help determine the size of the pending trade.

The code below returns a risk number (actually, a measure of volatility) based on the Average True Range (ATR) of the past 14 days of prices for the current instrument. Typically, but not always, ATR is a fractional number less than 1 (especially in times of low volatility).

public class

**ATR\_RiskManager : RiskManager {**

private int length = 14; // length of avg true range calculation

public override double **GetPositionRisk () {**

// if there are enough bars to calculate an ATR, use the ATR if (Bar.Count > length) return ATR.Value (Bar, length); else return 1;

}

}

Two other examples of risk manager (trade sizing) logic might be “For opening new positions, use 100 shares” and “For orders that close existing positions, use the existing position size.”

## 5.8 Exposure Manager Component

Exposure components consider the possible effects of a pending trade on a position or portfolio, and either accept or reject the proposed trade. They do not modify the signal in any way. Instead, they do their analysis of the situation, and simply say to a suggested trade, “yes” or “no.” If a signal is accepted, the signal proceeds to the next step in the signal flow. If not accepted, the signal is blocked so that no trade occurs.

Here is the code for the default strategy exposure manager. By default, the component contains a **Validate** method that is called by the framework to get a Yes or No answer. The Validate method must return a true or false to accept (true) or reject (false) the signal.

The default code shown below has been modified to print out a runtime message so you can see which component gets called first at strategy runtime. (The strategy component gets called before the metastrategy exposure component.) Of course, instead of the text message shown here, you would normally add code to check your portfolio against your equity and position exposure rules.

// Strategy level default exposure method public override bool

**Validate (Signal signal) {**

Console.WriteLine ("Hello from strategy Exposure component"); return base.Validate (signal);

}

## 5.9 MetaStrategy Components

This section describes the purpose and default implementations of metastrategy components.

Only the components that you would normally modify are discussed below. In particular, the Execution, Optimization, and Report components are not discussed here, because the default implementations provided by the framework are usually sufficient.

#### 5.9.1 Simulation Component

Here is some example code that shows how to specify what kinds of market data to feed into the metastrategy. For example, the code below feeds both 60-second and daily bars into the metastrategy.

public class

Daily\_Simulation : SimulationManager {

public override void

**Init () {**

// the name “Bar.Time.60” tells the simulated data provider to // send 60-second (one minute) bars to the strategies.

//SendMarketDataRequest("Trade");

//SendMarketDataRequest("Quote");

SendMarketDataRequest("Bar.Time.60");

SendMarketDataRequest("Bar.Time.300"); // 5 minute bars

SendMarketDataRequest("Bar.Time.600"); // 1 hour bars

SendMarketDataRequest ("Daily"); // daily bars

}

The simulation component only specifies the kind of market data that the metastrategy wants to see. The simulation component does not specify particular instruments to trade—that’s the job of the MarketManager component within each individual strategy.

To see how instrument names (Market Manager) and market data series types (Simulation Manger) are combined to specify a complete data series name, take an instrument name (e.g. MSFT), and prepend it to a market data type string such as “Bar.Time.60” or “Daily.” You end up with something like “MSFT.Bar.Time.60” or “MSFT.Daily”, which are the same names that appear in the IDE database or DataCenter for known series names.

#### 5.9.2 MetaStrategy Money Component

The metastrategy Money component is intended to allocate money among individual strategies according to an algorithm of your choice. This component is called automatically on startup, so that it can allocate funds to individual strategies before the strategies are started.

Here is the code for the default MetaMoneyManager component, with a line of code added to print out a console message for debugging. The Allocate method is called automatically by the framework. The default behavior is to allocate money evenly among all strategies, according to a formula like “amount = MetaCash / number of strategies”. But you can write your own code in the Allocate method to use a more sophisticated allocation algorithm.

public class

Dummy\_Money : MetaMoneyManager {

public override void

**Init () {**

// Add your code here

}

public override void **Allocate () {**

Console.WriteLine ("MetaStrategy Money");

base.Allocate();

}

#### 5.9.3 Metastrategy Risk Component

The metastrategy risk component is intended to control risk at the metastrategy level. When you create a new MetaRiskManager component, it contains several special methods and events.

Here is the default code generated for a new MetaRiskManager component.

public class

Dummy\_Risk : MetaRiskManager {

public override void

**Init () {**

// Add your code here

}

public override double **GetPortfolioRisk () {**  return 1;

}

public override void **OnMetaStrategyStarted () {**

// Add your code here

}

public override void

**OnStrategyPortfolioValueChanged (Strategy strategy) {**

// Add your code here

}

}

The GetPortfolioRisk method is a helper method for calculating the risk of the overall portfolio at the metastrategy level. It operates like the GetPositionRisk (note “Position” versus “Portfolio”) method at the strategy level, but works with the portfolio, rather than with a single position.

The OnMetaStrategyStarted event handler is fired when the metastrategy is first started up, in case you want to allocate risk or equity among individual strategies before they are started.

The OnStrategyPortfolioValueChanged event handler is called every time the value of a portfolio for a particular strategy is changed by an incoming price from a new trade. Each individual strategy has its own portfolio. So if the portfolio value for a particular trading strategy starts to drop too much, this event handler can identify the strategy and take action to stop the losses (perhaps by restricting the trading equity allocated to the strategy).

#### 5.9.4 MetaStrategy Exposure Component

The partitioned framework provides two levels of exposure control—one at the strategy level, and one at the portfolio level. Both components receive the original trading Signal object as input, and both return either true (to accept the signal and trade) or false (to reject the signal and trade).

At the strategy level, a typical exposure rule might be “Block any big trades over $10,000.” You might, for example, consider putting this kind of a rule in your strategy exposure manager when you are testing out new trading strategies, to catch situations when the strategy goes wild and tries to make big trades. You could also log such attempts from the strategy level exposure manager.

At the metastrategy level, a typical exposure rule might be “Block any trades from any concurrently executing strategies that would cause more than 10% of portfolio equity to be invested.” You might consider a rule like this when you first start doing live trading, for example, to restrict the amount of equity that can be invested.

Here is the code for the default metastrategy exposure manager. By default, the component contains a **Validate** method that is called by the framework to get a true or false answer. The Validate method must return a Boolean value to accept (true) or reject (false) the signal.

The default code shown below has been modified to print out a runtime message so you can see which component gets called first at strategy runtime. (The strategy component gets called before the metastrategy exposure component.) Of course, instead of the text message shown here, you would normally add code to check your portfolio against your equity and position exposure rules.

// MetaStrategy level default exposure method public override bool

**Validate (Signal signal) {**

Console.WriteLine ("Hello from MetaStrategy Exposure component"); return base.Validate (signal);

}

# 6 SmartQuant Example Strategies

This section provides documented code for several different kinds of trading strategies. Some are strategies that ship with the SmartQuant system. One is from Chande’s book *Beyond Technical Analysis*, and most others are from Altucher’s book *Trade Like a Hedge Fund.*

## 6.1 Summary of Strategy Techniques by Strategy

For your convenience, here is a cross reference between some useful strategy coding techniques and the strategy examples that follow. The associations below tell you which strategies use which strategy techniques to implement the strategy.

Techniques are usually only mentioned in the first strategy that uses the technique, even if the technique is used again by other strategies.

#### 6.1.1 5% Panic Recovery

* Uses OnBar
* Define a strategy variables as a user-settable parameter
* Test for an open position (if HasPosition)
* Calculate a price 5% below the current bar
* Send a limit order
* Send a market order
* Cancel a buy order

#### 6.1.2 Four Days Down and Long

* Count number of down days
* Discusses how bars and orders and times are related

#### 6.1.3 Four Days Up and Short for 1% Profit

* Uses OnBar, OnPositionOpened
* Use of a profit target 1%
* Use limit order for a profit target

#### 6.1.4 Breakout with 4% Entry Limit

* Uses OnBar, OnBarOpen (to catch leading edge of bar)
* Calculate 4% above previous closing price

#### 6.1.5 Breakout with Multiple Exits

* Uses OnBar, OnPositionOpened, OnPositionClosed
* Uses OnStopExecuted
* Uses look back channel of 30 bars (find highest high in a series)
* Uses ATSStop as internal stop reminder
* Uses OCA One Cancels All order groups • Exit after N bars have elapsed
* Exit after a profit target has been reached
* Exit when a trailing stop is executed

#### 6.1.6 Bollinger Bands

* Uses Bollinger Bands for range trading
* Uses SMA simple moving average
* Uses Series.Contains(datetime) to check for enough bars

#### 6.1.7 Bollinger Bands with Profit Target

* Uses penetration level parameter for strength of band crossing
* Exits on profit target
* Exit after 4 days

#### 6.1.8 Simple Moving Average Crossover

* Uses two SMA moving averages
* Uses ECross objects to detect crossovers
* Exits on moving average crossover
* Exits on internal trailing stop *indicator* (ATSStop)
* Exits on external trailing stop order (resides on exchange servers)

#### 6.1.9 Slow Turtle Trend Following

* Trades only on Monday mornings
* Uses very long averages (100 and 385 days)
* Reverses positions by trading 2 times current position size

#### 6.1.10 Chande’s 65sma\_3cc Crossover Strategy

* Uses OnBar, OnBarOpened, OnStopExecuted, OnPositionChanged
* Uses 3 consecutive closes above SMA for confirmation of price move
* Uses ADX indicator (Average Directional Index)
* Uses RAVI indicator (Range Action Verification Index)
* Uses Position.Side to test for position type (long or short)
* Uses helper methods to manage exits, positions
* Exits on moving average crossover
* Exits after N bars
* Exits on internal trailing stop

#### 6.1.11 Stock 2% Gap

* Enters when prices gap down 2%
* Exits (issues the order) on the very next bar

#### 6.1.12 Stock Down, Stock 2% Gap

• Checks for a down day previous to the 2% gap down day

#### 6.1.13 QQQQ Gap Down 0.5%, Stock Down, Stock Gap 5%

* Works with two instruments at once
* Coordinates several entry conditions among two instruments

#### 6.1.14 QQQQ Gap 0.5%, Stock Down, Stock Gap 5%, Hold Overnight

• Holds a position overnight (one more bar, for daily bars)

#### 6.1.15 Unilateral Pairs Trading

* Uses OnBarSlice to manage two instruments at once
* Shows how to make calculations for a volatility trade
* Uses many series of prices, ratios, differences, and SMAs of them all
* Uses SMD (standard deviation series)

#### 6.1.16 QQQ Crash, QQQQ Trade

* Uses Bollinger bands
* Catches drastic price moves (1.5 std dev) using a short moving average (10 days)

#### 6.1.17 QQQ Crash, Stock Trade

* Uses multiple instruments simultaneously
* Trades a volatile index stock when the index itself crashes

#### 6.1.18 Bollinger Bands with 5-Minute Bars

* Uses OnTrade to look at trade data during bar formation
* Uses OnPositionValueChanged to monitor profit for profit target
* Uses OnPositionOpened to SetStop to set a time interval stop (not a price stop)
* Uses OnStopExecuted to set an exit flag after N minutes have elapsed
* Intended for quick intraday trades of 5 or 10 minutes in length
* Enters on a 3% drop
* Exits on a profit target of 1%
* Exits after 2 bars have passed
* Uses flags to control trade entry and exit

## 6.2 Pattern Matching Strategies

A pattern matching strategy is one that looks for special patterns in the incoming market data, and tries to make profitable trades on those special patterns. We start our strategy examples with pattern matching strategies, because they provide a gentle introduction to syntax.

#### 6.2.1 5% Down-In-One-Day Panic Recovery

This first strategy is really simple, so it has been placed first in the document. The concept behind this strategy is that sometimes the market panics over some news, and hammers down a stock price for a short time. After a few minutes or hours, people recover from their panic, realize that the stock price is too low, and buy long to bid the price back up.

At the end of each trading day (we assume daily bars here, like Altucher did in his book), this strategy receives the daily bar in the OnBar event handler. If no position is open, the strategy uses today’s closing price to calculate a new limit price that is 5% lower than today’s close, and issues a limit buy order to be executed at market open the following day.

In contrast, if a position is open at the end of the day, it means that a 5% gap happened sometime during today, and so the strategy immediately closes the position with a market order (which will be executed at market open the next morning). The thinking here is that the panic prices would have been corrected upward during the day to something more reasonable.

Notice how the timing of strategy decisions and actions is affected by when the bars arrive, and when the orders are executed. With daily bars, the strategy receives the daily bar at the end of each trading day, in the OnBar event handler. The OnBar event handler is executed on the trailing edge of each bar, at time that corresponds to the end of the trading day for daily bars. If you want to do something on the leading edge of a bar—meaning at the market open before the daily bar is constructed—then you should put your code in an OnBarOpen event handler.

using System; using System.Drawing; using System.ComponentModel; using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{db87cd53-b704-44cb-9480-d3d56c754ab7}",

ComponentType.ATSComponent,

Name = "Tech13\_System1\_ATS",

Description = "5% down Panic Recovery")] public class

**Tech13\_System1\_ATS : ATSComponent {**

// buy when a stock drops 5% in one day private double percent = 5;

// orders and trade quantity private SingleOrder buyOrder; private double qty = 100;

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

public override void **OnBar (Bar bar) {**

// if we do not have a position, update the limit buy

// order to be 5% above today's close if (!HasPosition) {

// cancel the old limit order (it's out of date now) if (buyOrder != null) buyOrder.Cancel ();

// issue a new buy order at 5% below today's close

// this order will execute tomorrow if price is matched double buy\_price = bar.Close \* (1 - (percent / 100)); buyOrder = LimitOrder (Side.Buy, qty, buy\_price); buyOrder.Text = "Tech13\_System1 - Buy"; buyOrder.Send ();

}

// else we opened a position today using our limit

// order from yesterday, so now close the position at

// the end of today. We expect that such a big drop

// was freaky, and that prices recovered during the

// day. If not, this order stops further losses. else {

SingleOrder sellOrder = MarketOrder (Side.Sell, qty); sellOrder.Text = "Tech13\_System1 - Sell"; sellOrder.Send ();

}

}

}

#### 6.2.2 Four Down Days and Long

The concept of this strategy is to open a long position after a major market index has had 4 down days in a row. The theory is that 4 days of market momentum in a major market index is hard to maintain, and that an up day is soon to follow. You might have to wait a day or two for it to show up, but it will show up eventually.

This implementation buys at the market open on day 5 after 4 down days, and doesn’t wait around for multiple days for an up day. It just automatically issues a market sell order at the end of day 6 (when day 6 bar is received). The sell order will actually be executed by the exchange when the market opens on day 7.

Notice how the timing of strategy decisions and actions is affected by when the bars arrive, and when the orders are executed. With daily bars, the strategy receives the daily bar at the end of each trading day, in the OnBar event handler. The OnBar event handler is executed on the trailing edge of each bar, at time that corresponds to the end of the trading day for daily bars. If you want to do something on the leading edge of a bar—meaning at the market open before the daily bar is constructed—then you should put your code in an OnBarOpen event handler.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{ba049d57-cffc-46df-91ee-68e0ada68336}",

ComponentType.ATSComponent,

Name = "Tech17\_System1\_ATS",

Description = "Four Down Days")] public class

**Tech17\_System1\_ATS : ATSComponent {**

// count of down days private int count; private double prevClose;

// amount to trade private double qty = 100;

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

public override void

**Init () {**

prevClose = -1; count = 0;

}

public override void **OnBar (Bar bar) {**

// we need at least one bar to go by before we do anything if (prevClose == -1) {

// remember the first bar and return prevClose = bar.Close; return;

}

// if we don't have a position open, increment the count

// of down days (or reset it to zero) if (!HasPosition) { if (prevClose > bar.Close) count++; else count = 0;

// if this is the fourth down day, issue a market order // to open a long position tomorrow morning, on day 5 if (count == 4) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech17\_System2 - Buy"; order.Send ();

}

}

// else if we have a position open, close it now

// (today is the day after the trade was entered), // so “now” is actually end of day 6, and the trade // will be executed by the market on open day 7.

// All this discussion assumes daily bars, like the author did else {

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech17\_System2 - Sell"; order.Send ();

}

// today's close now becomes the previous close for the

// down day calculation prevClose = bar.Close;

}

}

#### 6.2.3 Four Up Days and Short for 1% Profit

This strategy is a variation on the Four Down Days strategy. A plain Four Up Days (flip) of the down days strategy did not produce good results for Altucher, reinforcing his statement that going short is not the exact opposite of going long. So he provides this “up day” variation, which uses up days instead of down days, but it also enforces two extra conditions.

The first condition is that the fourth day up has to be a huge one (2% or more). After 3 days up already, going up 2% on the fourth day would be quite impressive. Theoretically, such a jump on the fourth day would be a strong invitation to long stock holders to take some profits off the table the next morning, thereby driving down the price. So this strategy issues a short order on the close of the fourth day, for execution at the open of the fifth day.

The second condition is that the strategy exits when it reaches a 1% profit target, and uses a limit order to implement the exit during trading day 5. If the position is still open at the end of day 5, the strategy issues a market order to close the position, for execution at market open on day 6. Keep in mind that this discussion assumes the use of daily bars (as the book author did).

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{429bc1f7-519f-425d-864f-892e54428958}",

ComponentType.ATSComponent,

Name = "Tech17\_System3\_ATS",

Description = "Four Up Days And Short for 1%")] public class

**Tech17\_System3\_ATS : ATSComponent {**

// count of up days private int count; private double prevClose;

// exit with a profit target of 1% private double profitTarget = 1; // last day up must be a big 2% up day private double upPercent = 2;

// orders and quantity private SingleOrder coverOrder; private SingleOrder shortOrder; private double qty = 100;

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Profit Target")] public double ProfitTarget { get { return profitTarget; } set { profitTarget = value; }

}

[Category ("Parameter"), Description ("Up Move (Percent)")] public double UpPercent { get { return upPercent; } set { upPercent = value; }

}

public override void

**Init () {**

prevClose = -1; count = 0;

}

public override void **OnBar (Bar bar) {**

// we need to let a bar go by to capture the prev close if (prevClose != -1) {

// if we don't have a position open, update the count

// of up days, and try to enter a trade if (!HasPosition) { if (prevClose < bar.Close) count++; else count = 0;

// if we have seen 4 up days, AND if the last day

// up was 2% or more, then open a new position,

// going short on the day's close if (count == 4) {

if ((bar.Close - prevClose)

/ prevClose >= upPercent / 100) {

shortOrder = MarketOrder (Side.Sell, qty); shortOrder.Text = "Tech17\_System3 - Sell"; shortOrder.Send ();

}

}

}

// if we have a position open, cancel our previous

// 1% profit target order, and close using a market order else {

coverOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech17\_System3 - Buy"; order.Send ();

}

}

// now today's close becomes the previous close prevClose = bar.Close;

}

public override void **OnPositionOpened () {**

// when we open a position, immediately issue a limit order

// for our 1% profit target

double target\_price = shortOrder.AvgPx \* (1 - profitTarget / 100); coverOrder = LimitOrder (Side.Buy, qty, target\_price); coverOrder.Text = "Tech17\_System3 - Buy"; coverOrder.Send ();

}

}

## 6.3 Breakout Strategies

The concept of a breakout strategy is to monitor prices over a period of time (called the look back time, or the breakout channel length) so that you can determine a trading range for past prices.

Then when a new price move suddenly “breaks out” of the usual trading channel, it could be the beginning of a big new price move upward.

So when a breakout strategy sees current prices break out of the usual trading channel, it opens a trading position and holds it until some exit criterion is satisfied. Many exit criteria are possible— you can exit after N bars go by, after a profit target has been reached, or after the trade goes against you and your stop loss order is executed.

#### 6.3.1 Breakout with 4% Entry Limit

The rationale behind this simple breakout strategy is to take advantage of a big (4%) daily move upward that would likely squeeze all those short sellers who were hoping the stock would go down. This strategy thinks that moving 4% up in a single day is such a big move that it would scare all the short sellers into covering their positions by buying long, thereby driving the price up even further the next day. (Of course, contrarian traders would be thinking of going short the next day, to take advantage of a price recovery to lower levels.)

This strategy uses the OnBarOpen event handler, so that all trading events take place at the beginning of the day, at market open (on the leading edge of the daily bar).

As each day begins, the strategy calculates an entry price 4% higher than yesterday’s close, and issues a limit order to open a long position if that price is reached. The next morning, if a position exits (opened during yesterday’s trading), the strategy issues a market order to close the position.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{001ab71e-8794-4dee-8c5c-5f7c158f5fcb}",

ComponentType.ATSComponent,

Name = "Tech17\_System4\_ATS",

Description = "The 4 Percent Breakout Move")] public class

**Tech17\_System4\_ATS : ATSComponent {**

// enter trade on 4 breakoutPercent breakout private double breakoutPercent = 4;

// orders and trade quantity private SingleOrder buyOrder; private int qty = 100;

// for calculating breakout price limit private double prevClose;

[Category ("Parameter"), Description ("Quantity")] public int Qty {

get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return breakoutPercent; } set { breakoutPercent = value; }

}

public override void

**Init () {**

prevClose = -1;

}

public override void **OnBarOpen (Bar bar) {**

// we need to let the first bar go by before we can

// calculate the breakout limit if (prevClose != -1) {

// if we do not have a position, then cancel the

// previous limit order (it is out of date) if (!HasPosition) { if (buyOrder != null) buyOrder.Cancel ();

// now try to enter a trade by setting a limit order // to automatically buy in if the big 4% jump arrives.

// This order will reside on the exchange servers, and

// will execute during the day if the limit is triggered. double breakout\_fraction = 1 + (breakoutPercent / 100); double breakout\_price = prevClose \* breakout\_fraction; buyOrder = StopOrder (Side.Buy, qty, breakout\_price); buyOrder.Text = "Tech17\_System4 - Buy"; buyOrder.Send ();

}

// if we have a position open, then close it now.

// Now (which is the leading edge of today's daily bar)

// is the start of the day after the trade was opened. else {

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech17\_System3 - Sell"; order.Send ();

}

}

}

public override void **OnBar (Bar bar) {**

// update the prevClose value for the breakout calculation prevClose = bar.Close;

}

}

#### 6.3.2 Breakout with Multiple Exits

The following example implements a simple breakout strategy using an integrated framework. (As you will see, all the examples use the integrated framework because the code is so much easier to write.) By default, this strategy looks back 30 bars to detect breakouts, and exits when one of its three possible exit criteria is satisfied. It can exit after 10 bars have gone by. It can exit when a profit target has been reached. And it can exit when a trailing stop limit is reached.

Notice the use of a stop indicator, rather than a stop order, to exit the trade with a trailing stop. The stop is just a stop that is maintained by the strategy framework (not the broker). The trailing stop fires when an incoming trade price reaches the stop limit, and the OnStopExecuted event handler is called. The event handler issues a market order to close the position. This way, no stop order is ever issued to the broker (perhaps some brokers or exchanges don’t take trailing stops).

using System; using System.Drawing; using System.ComponentModel; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.Series; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution; using SmartQuant.Trading; using SmartQuant.Providers;

[StrategyComponent ("{0f2cec85-e5f3-4314-b396-25a4e1268bff}",

ComponentType.ATSComponent,

Name = "Breakout\_ATS",

Description = "SmartQuant Example")] public class

**Breakout\_ATS : ATSComponent {**

// a series to store bars that form the breakout channel

// By default we want to look back 30 days (bars) BarSeries series;

int length = 30;

// set stop loss at 98% of today's closing price

// which is a stop loss of 2% below today's close double stopOCALevel = 0.98;

// set profit target to 5% above today's close double limitOCALevel = 1.05; int OCACount = 0;

// stop order parameters ATSStop stop;

double stopLevel = 0.05;

StopType stopType = StopType.Trailing; StopMode stopMode = StopMode.Percent;

// entry and exit control flags // we have three ways of exiting here, controlled

// by the Properties panel bool entryEnabled = true;

// only use one exit method at a time bool ocaExitEnabled = true; bool timeExitEnabled = false; bool stopExitEnabled = false; // orders we use for this strategy

SingleOrder marketOrder;

SingleOrder limitOrder;

SingleOrder stopOrder;

// buy 100 shares at a time double qty = 100;

// exit breakout trades after 10 bars go by int barCount; int barsToExit = 10;

public

Breakout\_ATS () : base () { }

[Category ("Parameters")] public int Length { get { return length; } set { length = value; }

}

[Category ("OCA")] public double StopOCALevel { get { return stopOCALevel; } set { stopOCALevel = value; }

}

[Category ("OCA")] public double LimitOCALevel { get { return limitOCALevel; } set { limitOCALevel = value; }

}

[Category ("Exit")] public bool OCAExitEnabled { get { return ocaExitEnabled; } set { ocaExitEnabled = value; }

}

[Category ("Exit")] public bool TimeExitEnabled { get { return timeExitEnabled; } set { timeExitEnabled = value; }

}

[Category ("Exit")] public bool StopExitEnabled { get { return stopExitEnabled; } set { stopExitEnabled = value; }

}

[Category ("Exit")] public int BarsToExit { get { return barsToExit; } set { barsToExit = value; }

}

[Category ("Stop")] public double StopLevel { get { return stopLevel; } set { stopLevel = value; }

}

[Category ("Stop")] public StopType StopType { get { return stopType; } set { stopType = value; }

}

[Category ("Stop")] public StopMode StopMode { get { return stopMode; } set { stopMode = value; }

}

[Category ("Money")] public double Qty { get { return qty; } set { qty = value; }

}

public override void

**Init () {**

// attach a new series to store bars series = new BarSeries ();

}

public override void **OnBar (Bar bar) {**

// entry is disabled if we have an existing trade in play if (entryEnabled) {

// don't trade until we accumulate enough bars in our series

// for a proper look back of "length" bars into the past

// channel that we are trying to break out of if (series.Count > length)

// if today's high is higher than all bars in the

// lookback period of length "length" (30 by default) if (bar.High > series.HighestHigh (length)) {

// then we are breaking out long, so issue a long

// side market buy order to open a position marketOrder = MarketOrder (Instrument, Side.Buy, qty); marketOrder.Text = "Entry"; marketOrder.Send ();

// if one cancels all exit method is desired, we

// also issue a limit (profit target) order, and // a stop loss order in case the breakout fails.

// The OCA exit method uses a real stop loss order.

// The Stop exit method uses a stop indicator. // Use either the OCA or Stop method, not both at once.

if (ocaExitEnabled) {

// create and send a profit limit order double profitTarget = limitOCALevel \* bar.Close;

limitOrder = LimitOrder (Instrument, Side.Sell, qty,

profitTarget);

limitOrder.OCAGroup = "OCA " + Instrument.Symbol + " "

+ OCACount;

limitOrder.Text = "Limit OCA " + OCACount; limitOrder.Send ();

// create and send a stop loss order double lossTarget = stopOCALevel \* bar.Close;

stopOrder = StopOrder (Instrument, Side.Sell, qty,

profitTarget);

stopOrder.OCAGroup = "OCA " + Instrument.Symbol + " "

+ OCACount;

stopOrder.Text = "Limit OCA " + OCACount; stopOrder.Send ();

// bump the OCA count to make OCA group strings unique OCACount++;

}

// prevent further entries since we have entered already entryEnabled = false; barCount = 0;

}

}

// if entry is disabled on this bar, we have an open position else {

// increment bar count seen while position is open

// this count is used for the bar count exit method barCount++;

// if we want to exit the trade based on time, we use the

// bar count as a proxy for elapsed time, and issue a market

// sell order. Only one of OCA, Stop, or Time exit methods // should be enabled at the same time.

if (timeExitEnabled && (barCount > barsToExit)) { marketOrder = MarketOrder (Instrument, Side.Sell, qty); marketOrder.Text = "Time Exit"; marketOrder.Send ();

// also enable entries, since we closed our position entryEnabled = true;

}

}

// store the current bar in our series array, so we can look

// back to spot breakouts from past bars series.Add (bar);

}

public override void **OnPositionOpened () {**

// if we want to exit trades using the Stop method, set a

// a trailing stop indicator when the position is

// first opened. The stop indicator is not a stop loss // order that can be executed by a broker. Instead, the stop // just fires the OnStopExecuted event when it it triggered. if (stopExitEnabled)

stop = SetStop (stopLevel, stopType, stopMode);

}

public override void **OnPositionClosed () {**

// when a position is closed, cancel the limit and stop

// orders that might be associated with this position.

// But only cancel if the order has not been filled or

// not been cancelled already.

if (ocaExitEnabled &&

!(limitOrder.IsFilled || limitOrder.IsCancelled)) {

limitOrder.Cancel ();

}

// allow entries once again, since our position is closed entryEnabled = true;

}

public override void **OnStopExecuted (ATSStop stop) {**

// if our trailing stop indicator was executed, // issue a market sell order to close the position. marketOrder = MarketOrder (Instrument, Side.Sell, qty); marketOrder.Text = "Stop Exit"; marketOrder.Send ();

}

}

Of course, the breakout strategy shown above could be modified to use different channel lengths, different means of calculating the breakout channel limits (e.g. if prices went outside Bollinger bands, or if a particular volume was required on the breakout), and could use different means of maintaining or exiting the trade.

#### 6.3.3 Confirmation Methods

When a potential trade signal is at hand, such as prices breaking out of a channel, some strategies prefer to “confirm” the breakout action by checking for additional conditions that must be satisfied before entering a trade. These conditions are not part of the breakout signal itself, but instead are confirmations obtained from other sources that are different than the source of the breakout signal itself.

For example, suppose a breakout signal was defined as in the example above, such that a bar with a High > the highest High of the past 30 bars met the definition of a breakout signal. How could a strategy further verify that the breakout was a quality breakout by checking other conditions?

One possible confirmation could be to require that the current High exceed the highest High in the look back period by a particular amount. More of a gap could imply more probability of success.

Another method might be to require that the current bar Close be higher than the highest High. Yet another method might be to require 3 more consecutive closes that are each higher than the one before, where the first one is the breakout signal. This “3cc” method is used in a strategy example later in this document.

Still another method might be to check several other technical indicators, moving averages, or other sources of information to detect and filter out low quality breakouts. Many different kinds of confirmation methods are possible, limited only by your imagination.

## 6.4 Range Trading Strategies

When instrument prices trade sideways within a range of prices, we say that the prices are “in a trading range.” We can model the range technically using trading *bands* or *envelopes* of various kinds to model the upper and lower limits of the price trading range.

Most band and envelope models use a central moving average to represent the center of the trading range. Obviously the moving average can drift upward or downward to some extent as it follows prices. If it drifts up or down too much, we would say that prices are trending rather than ranging, but that doesn’t matter for the purposes of this discussion.

The outer bands are commonly calculated using various mixes of percentages of the moving average price, the standard deviation of the price series, or the highest high and lowest low of the past N bars in the series. Each of these methods has advantages and disadvantages.

As a general rule, long range trading strategies try to buy at the low end of the range, and sell at the middle (moving average line) or high end (upper band limit). Short range trading strategies try to do the opposite—sell high first, and then buy back when prices move back to the center line.

#### 6.4.1 Bollinger Bands

Bollinger bands are calculated using the standard deviation of the difference between prices and a central moving average line. Since Bollinger bands use standard deviations, they move apart during volatile price times, and get narrower during quiet price movement times.

Since range trading is basically a volatility technique (you trade edges against a center line), it works well in sideways markets where your open trades are not exposed to very much trending price action. That is, in sideways markets, there is a lower probability that your trade will go against you. But be aware, if a breakout occurs against your range trade, you can lose lots of money if you don’t have a stop loss in place.

The implementation shown here does not use stop losses, so it has no downside protection.

In the code that follows, the strategy tries to trade long by buying at the lower Bollinger band limit and selling when prices go up and reach the moving average line. The strategy constantly watches prices (bars), so it can update both the buy point and sell point as prices move around.

The strategy constantly updates the buy limit order at the broker while no position is open. When a buy occurs, the OnPositionOpened event handler is called to issue a sell limit order. From that point on, the strategy updates only the exit limit order, until the sell point is reached and the position is closed. Then the cycle repeats.

This strategy is effective in sideways markets and upward trending markets, if false breakouts and down trending markets don’t go against you too often. Of course you should add a stop loss mechanism to this strategy for downside protection before using it in real life. Also pay close attention to transaction costs, because small price moves don’t usually provide much profit.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{3362fbfa-b3db-48b8-8b44-e4e06475461a}",

ComponentType.ATSComponent, Name = "Tech5\_System1\_ATS",

Description = "Bollinger Band System1")] public class

**Tech5\_System1\_ATS : ATSComponent {**

// for bollinger band series private BBL bbl; private double bblOrder = 2;

// for simple moving average private SMA sma; private int smaLength = 20; // order quantity and types private double qty = 100; private SingleOrder buyOrder; private SingleOrder sellOrder;

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Order of BBL")] public double BBLOrder { get { return bblOrder; } set { bblOrder = value; }

}

[Category ("Parameter"), Description ("Length of SMA")]

[OptimizationParameter (5, 20, 1)] public int SMALength { get { return smaLength; } set { smaLength = value; }

}

public override void

**Init () {**

// get reference to bar series for current instrument

// Instrument is a property of the base class

BarSeries bars = Bars[Instrument]; // set up the moving averages sma = new SMA (bars, smaLength); sma.Color = Color.Green; Draw (sma, 0);

// set up bollinger bands bbl = new BBL (bars, smaLength, bblOrder);

bbl.Color = Color.Blue;

Draw (bbl, 0);

}

public override void **OnBar (Bar bar) {**

// always a good practice to be sure a series contains

// a bar for a particular date before you try to use it if (bbl.Contains (bar.DateTime)) {

// We are always trying to buy at the lower Bollinger

// limit, and sell when the price goes up to the

// latest SMA value. So we are constantly // updating both the buy point and the sell point.

// if we don't have an open position in this instrument,

// update the buy point to the latest lower bbl limit if (!HasPosition) { if (buyOrder != null) buyOrder.Cancel ();

buyOrder = LimitOrder (Side.Buy, qty, bbl.Last); buyOrder.Text = "Tech5\_System1 - Buy"; buyOrder.Send ();

}

// else if we already have a position going, update

// the sell point to follow the latest SMA value else {

UpdateExitLimit ();

}

}

}

public override void **OnPositionOpened () {**

UpdateExitLimit ();

}

private void **UpdateExitLimit () {**

// cancel old exit point order, if it exists if (sellOrder != null) sellOrder.Cancel (); // Issue a new order with the latest SMA value. This is

// kind of a "trailing SMA sell order" that follows the SMA.

sellOrder = LimitOrder (Side.Sell, qty, sma.Last); sellOrder.Text = "Tech5\_System1 - Sell"; sellOrder.Send ();

}

}

#### 6.4.2 Bollinger Bands with Profit Target

This example is like the basic Bollinger Band example above, but it adds three refinements. First, it demands that prices really punch through the lower Bollinger limit before buying. If 0 is the low limit, and 100 the high limit, this strategy buys only if prices are 20% of the band width below the lower limit. Second, it exits when a profit target of 15% is reached. Third, it exits after 4 days if the profit target has not been reached.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{51b48698-5012-44f9-abe3-4741e07a33bf}",

ComponentType.ATSComponent, Name = "Tech5\_System2\_ATS",

Description = "Bollinger Band with Profit Target")] public class

**Tech5\_System2\_ATS : ATSComponent {**

// B is a measure of how decisively prices break the

// bands. B is negative below the lower band, 0 at

// the lower band, 100 at the high band, and

// > 100 higher than the high band.

private B b;

// buy long when b is 20 below the low band private double bLevel = -20;

// 10 day moving average, 1.5 std deviation bands private int bLength = 10; private double bOrder = 1.5;

// we exit at a profit target private double profitPercent = 15;

// we exit after 4 days (bars) private int maxDuration = 4; private DateTime entryDate; private DateTime exitDate; private double exitPrice; private bool waiting;

// orders and quantity private double qty = 100; private SingleOrder sellOrder; private SingleOrder buyOrder;

[Category ("Parameter"), Description ("%b level to long")] public double BLevel { get { return bLevel; } set { bLevel = value; }

}

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent profit targe")] public double ProfitPercent { get { return profitPercent; } set { profitPercent = value; }

}

[Category ("Parameter"), Description ("Max number of days, while position is active")] public int MaxDuration { get { return maxDuration; } set { maxDuration = value; }

}

[Category ("Parameter"), Description ("Length of %B")] public int BLength { get { return bLength; } set { bLength = value; }

}

[Category ("Parameter"), Description ("Order of %B")] public double BOrder { get { return bOrder; } set { bOrder = value; }

}

public override void

**Init () {**

// get ref to bar series for this instrument

BarSeries bars = Bars[Instrument];

// set up bollinger bands

BBL bbl = new BBL (bars, bLength, bOrder); bbl.Color = Color.Blue;

Draw (bbl, 0);

// set up a series for B (breakout force) value b = new B (bars, bLength, bOrder);

b.Color = Color.Blue;

Draw (b, 2);

}

public override void **OnBar (Bar bar) {**

// good practice to check if a series has the date

// you are interested in before you try to use it if (b.Contains (bar.DateTime)) {

// if we don't have a position and prices are

// 20 below the lower band, open a long position if (!HasPosition) {

if (b[bar.DateTime] \* 100 <= bLevel) { buyOrder = MarketOrder (Side.Buy, qty); buyOrder.Text = "Tech5\_System2 - Buy"; buyOrder.Send ();

}

}

}

}

public override void **OnPositionOpened () {**

// when a position is opened, calculate profit target exitPrice = buyOrder.AvgPx \* (1 + profitPercent / 100);

// and calc N trading days left until the exit day entryDate = Clock.Now.Date; waiting = true; exitDate = entryDate.Date; int days = maxDuration + 1; while (days > 0) { exitDate = exitDate.AddDays (1); if (exitDate.DayOfWeek != DayOfWeek.Sunday && exitDate.DayOfWeek != DayOfWeek.Saturday) days--;

}

}

public override void **OnBarOpen (Bar bar) {**

// on the leading edge of each new bar (market open)

// if we have a position open, try to exit if (HasPosition) {

// if we have not reached the days limit,

// issue a sell limit order for the profit target if (waiting && bar.DateTime >= entryDate.AddDays (1)) { PlaceExit (); waiting = false;

}

// if we \_have\_ reached the exit day (4 days after entry),

// cancel the profit target sell order, and issue a new

// market order to close the position now.

if (bar.DateTime >= exitDate) { if (sellOrder != null) sellOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech5\_System2 - Sell (Max Duration)"; order.Send ();

}

}

} private void **PlaceExit () {**

// cancel existing sell order if there is one if (sellOrder != null) sellOrder.Cancel ();

// issue a new sell limit order at the profit target price sellOrder = LimitOrder (Side.Sell, qty, exitPrice); sellOrder.Text = "Tech5\_System2 - Sell (Profit Target)"; sellOrder.Send ();

}

}

## 6.5 Trend Following Strategies

Moving average strategies are probably the most well known of all computer trading strategies. Typically they are implemented with a goal of following a trend in price movement, so many moving average strategies are also trend following strategies. Notice that “trend following” is the goal of the strategy, and the implementation method is “moving average crossover.”

Of course moving average methods can be used with strategies whose goal is to NOT go with the trend. For example, maybe strategies want to use moving averages to go *against* the trend, or to use moving averages as confirmation signals, or exit-only signals. The main point here is that moving averages have many possible purposes beyond trend following.

#### 6.5.1 Simple Moving Average Crossover

The following moving average crossover strategy ships with the SmartQuant system, and uses moving average crossovers to enter trades. This strategy (like the breakout strategy shown above) has three methods for exiting a trade. It can use the moving averages, the OCA One Cancels All method, or a trailing stop indicator method (which initiates a market order to close the position).

Notice the use of a trailing stop indicator, rather than a stop order, to exit the trade. The stop is just a stop that is maintained by the strategy framework (not the broker). The trailing stop fires when an incoming trade price reaches the stop limit, and the OnStopExecuted event handler is called. The event handler issues a market order to close the position. This way, no stop order is ever issued to the broker (perhaps some brokers or exchanges don’t take trailing stops).

using System; using System.Windows.Forms; using System.Drawing; using System.ComponentModel;

using SmartQuant.FIX; using SmartQuant.Optimization; using SmartQuant.Data; using SmartQuant.Series; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution; using SmartQuant.Trading; using SmartQuant.Providers; [StrategyComponent ("{75445e02-c779-4420-a3cf-4943be3014b2}",

ComponentType.ATSComponent,

Name = "Crossover\_ATS", Description = "")]

public class

**Crossover\_ATS : ATSComponent {**

// this strategy uses some of the same exit methods

// as the breakout strategy described earlier. Look

// there for additional documentation

// lengths and colors of the simple moving averages int length1 = 3; int length2 = 7; Color color1 = Color.Red;

Color color2 = Color.Blue;

SMA sma1;

SMA sma2;

// only one trade is allowed at a time bool entryEnabled = true; // we have three ways to exit. Use only

// one exit method at a time. bool crossoverExitEnabled = true; bool ocaExitEnabled = true; bool stopExitEnabled = true; // parameters for the OCA exit method

// OCA method uses a real stop loss order double stopOCALevel = 0.98; double limitOCALevel = 1.05; int OCACount = 0; // parameters for trailing stop exit method // Stop method does not use a real stop order.

// It just uses an internal framework event. ATSStop stop; double stopLevel = 0.05; StopType stopType = StopType.Trailing;

StopMode stopMode = StopMode.Percent;

// for the orders used by this strategy

SingleOrder marketOrder, limitOrder, stopOrder;

// number of shares to buy double qty = 100;

[Category ("SMA")] public int Length1 { get { return length1; } set { length1 = value; }

}

[Category ("SMA")] public int Length2 { get { return length2; } set { length2 = value; }

}

[Category ("SMA")] public Color Color1 { get { return color1; } set { color1 = value; }

}

[Category ("SMA")] public Color Color2 { get { return color2; } set { color2 = value; }

}

[Category ("OCA")] public double StopOCALevel { get { return stopOCALevel; } set { stopOCALevel = value; }

}

[Category ("OCA")] public double LimitOCALevel { get { return limitOCALevel; } set { limitOCALevel = value; }

}

[Category ("Stop")] public double StopLevel { get { return stopLevel; } set { stopLevel = value; }

}

[Category ("Stop")] public StopType StopType { get { return stopType; } set { stopType = value; }

}

[Category ("Stop")] public StopMode StopMode { get { return stopMode; } set { stopMode = value; }

}

[Category ("Exit")] public bool CrossoverExitEnabled { get { return crossoverExitEnabled; } set { crossoverExitEnabled = value; }

}

[Category ("Exit")] public bool OCAExitEnabled { get { return ocaExitEnabled; } set { ocaExitEnabled = value; }

}

[Category ("Exit")] public bool StopExitEnabled { get { return stopExitEnabled; } set { stopExitEnabled = value; }

}

[Category ("Money")] public double Qty { get { return qty; } set { qty = value; }

} public

**Crossover\_ATS () : base () { }**

public override void

**Init () {**

// get reference to incoming bar series

BarSeries series = DataManager.Bars[Instrument]; // set up the moving averages, based on closing prices sma1 = new SMA (series, length1, BarData.Close, color1); sma2 = new SMA (series, length2, BarData.Close, color2); sma1.Color = Color.Green; sma2.Color = Color.Blue;

// 0 means draw both averages on the price chart

Draw (sma1, 0);

Draw (sma2, 0);

}

public override void **OnBar (Bar bar) {**

// does the fast average cross over the slow average?

// if so, time to buy long

ECross cross = sma1.Crosses (sma2, bar.DateTime); // we only allow one active position at a time if (entryEnabled) {

// if price trend is moving upward, open a long

// position using a market order, and send it in if (cross == ECross.Above) {

marketOrder = MarketOrder (Instrument, Side.Buy, qty); marketOrder.Text = "Entry"; marketOrder.Send ();

// if one cancels all exit method is desired, we

// also issue a limit (profit target) order, and // a stop loss order in case the breakout fails.

// The OCA exit method uses a real stop loss order.

// The Stop exit method uses a stop indicator. // Use either the OCA or Stop method, not both at once.

if (ocaExitEnabled) {

// create and send a profit limit order double profitTarget = limitOCALevel \* bar.Close;

limitOrder = LimitOrder (Instrument, Side.Sell, qty,

profitTarget);

limitOrder.OCAGroup = "OCA " + Instrument.Symbol + " "

+ OCACount;

limitOrder.Text = "Limit OCA " + OCACount; limitOrder.Send ();

// create and send a stop loss order

double lossTarget = stopOCALevel \* bar.Close;

stopOrder = StopOrder (Instrument, Side.Sell, qty,

lossTarget);

stopOrder.OCAGroup = "OCA " + Instrument.Symbol + " "

+ OCACount;

stopOrder.Text = "Stop OCA " + OCACount; stopOrder.Send ();

// bump the OCA count to make OCA group strings unique

OCACount++;

}

entryEnabled = false;

}

}

// else if entry is disabled on this bar, we have an open position else {

// if we are using the crossover exit, and if the fast

// average just crossed below the slow average, issue a // market order to close the existing position

if (crossoverExitEnabled) if (cross == ECross.Below) {

marketOrder = MarketOrder (Instrument, Side.Sell, qty); marketOrder.Text = "Crossover Exit"; marketOrder.Send ();

}

}

}

public override void **OnPositionOpened () {**

// if we want to exit trades using the Stop method, set a

// a trailing stop indicator when the position is

// first opened. The stop indicator is not a stop loss // order that can be executed by a broker. Instead, the stop // just fires the OnStopExecuted event when it it triggered. if (stopExitEnabled)

stop = SetStop (stopLevel, stopType, stopMode);

}

public override void **OnPositionClosed () {**

// when a position is closed, cancel the limit and stop

// orders that might be associated with this position.

// But only cancel if the order has not been filled or

// not been cancelled already.

if (ocaExitEnabled &&

!(limitOrder.IsFilled || limitOrder.IsCancelled)) {

limitOrder.Cancel ();

}

// allow entries once again, since our position is closed entryEnabled = true;

}

public override void

**OnStopExecuted (ATSStop stop) {**

// if our trailing stop indicator was executed, // issue a market sell order to close the position. marketOrder = MarketOrder (Instrument, Side.Sell, qty); marketOrder.Text = "Stop Exit"; marketOrder.Send ();

}

}

#### 6.5.2 “Slow Turtle” Trend-Following

The original Turtle system was made famous because it was supposed to show that anyone could become a good commodities trader—all you needed was to follow some specific rules that implemented a good trading system.

The implementation shown below is based on Altucher’s book, which uses its own choice of 22 *week* (100 day) and 55 *week* (385 day) moving averages. This is what he calls a “slow turtle” strategy that is intended to catch very long (and major) trends. He does this by using very long moving averages. The original Turtle system used shorter averages. (It is worth saying that there are many variations of the Turtle system on the Internet.)

There are three interesting things about this single-state (always in the market) implementation. It uses very long moving averages (100 and 385 days). It only trades on Monday mornings (in the OnBarOpen method). And it reverses existing positions by buying or selling a trade quantity that is twice the size of the existing position.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{d807c95d-a801-48c0-9330-04b9800dbd99}",

ComponentType.ATSComponent,

Name = "Tech7\_System\_ATS",

Description = "Altucher Version of the Turtle System")] public class

**Tech7\_System\_ATS : ATSComponent {**

// quantity to buy on a trade private double qty = 100;

// two moving averages private SMA fastSMA; private int fastSMALength = 22; private SMA slowSMA;

private int slowSMALength = 55;

private DateTime buyEntryDate = DateTime.MaxValue; private DateTime sellEntryDate = DateTime.MaxValue;

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Length of SMA in weeks")] public int FastSMALength { get { return fastSMALength; } set { fastSMALength = value; }

}

[Category ("Parameter"), Description ("Length of SMA in weeks")] public int SlowSMALength { get { return slowSMALength; } set { slowSMALength = value; }

}

public override void

**Init () {**

// get a reference to bar series for current instrument

BarSeries bars = Bars[Instrument];

// set up the fast average

fastSMA = new SMA (bars, fastSMALength \* 7);

fastSMA.Color = Color.Blue; Draw (fastSMA, 0);

// set up the slow average

slowSMA = new SMA (bars, slowSMALength \* 7);

slowSMA.Color = Color.Red;

Draw (slowSMA, 0);

}

public override void **OnBarOpen (Bar bar) {**

// calc quantity to reverse a position double orderQty = 2 \* qty; if (!HasPosition) orderQty = qty;

// The strategy always trades on Monday morning.

// if today is =Monday or past a Monday holiday

// open a new position long (or reverse an existing short pos) if (buyEntryDate <= bar.DateTime) {

MarketOrder order = MarketOrder (Side.Buy, orderQty); order.Text = "Tech7\_System - Buy"; order.Send ();

// reset the entry date to infinity.

// The entry date will be reset next time we need to buy,

// and the infinity will prevent this code from being executed

// again until we need to execute it buyEntryDate = DateTime.MaxValue;

}

// if today is =Monday or past a Monday holiday

// Open a short position (or reverse a long position) if (sellEntryDate <= bar.DateTime) {

MarketOrder order = MarketOrder (Side.SellShort, orderQty); order.Text = "Tech7\_System - Sell Short"; order.Send ();

// reset next sell date to infinity to // avoid executing this code until we need to sellEntryDate = DateTime.MaxValue;

}

}

public override void **OnBar (Bar bar) {**

// if our SMAs contain the current bar date if (fastSMA.Contains (bar.DateTime) && slowSMA.Contains (bar.DateTime)) {

// see which one is above the other

ECross cross = fastSMA.Crosses (slowSMA, bar.DateTime);

// the strategy always buys or sells on a Monday morning

// So this code finds the Monday, and the trade is executed

// in the OnBarOpen event handler on Monday morning

DayOfWeek dayofweek = (int)bar.DateTime.DayOfWeek; DateTime monday;

monday = bar.DateTime.AddDays (7 - (dayofweek - 1) % 7); if (cross == ECross.Above) buyEntryDate = monday; if (cross == ECross.Below) sellEntryDate = monday;

}

}

}

#### 6.5.3 Chande’s 65sma\_3cc Strategy

This strategy is a moving average crossover strategy from Chande’s *Beyond Technical Analysis* book. It uses a 65 day simple moving average (65sma), and an addition confirmation condition of “three consecutive closes” (3cc). Thus the name of this strategy is called “65sma\_3cc.”

Like other strategies shown previously, this strategy enters on a single condition (moving average crossover with 3 consecutive higher closes), but can use three possible exit methods. It can exit after a particular number of bars go by, it can exit on a trailing stop, or it can exit on a moving average crossover in the opposite direction. The moving average crossover exit is called the “trend following exit” in the code.

This is a more complex strategy than the ones previously shown, because it uses several new technical indicators to control trade entries and exits. In particular, to avoid trading whipsaws, this strategy uses two trade entry filters based on ADX and RAVI index values.

ADX stands for Average Directional Index. This is a technical indicator that helps to recognize trending markets, so that trend following systems can avoid being whipsawed by fast reversals in moving average crossovers in flat or sideways markets. When this index rises, it indicates a trending market; when it falls, a sideways market. Typically, when the ADX indicator is above 40, and then falls, a sideways or consolidating market is emerging. Conversely, when the indicator is below 20 and then rises, a trending market is emerging. The 65sma\_3cc strategy uses an ADX filter to avoid entering new trades in a sideways market.

RAVI stands for Range Action Verification Index. This is a technical indicator that also helps to recognize trending markets, for the same purpose of helping trend following systems to avoid entering whipsaw trades in sideways markets. The RAVI index is a moving average crossover system itself—it uses a 7-day fast average and a 65-day slow average. The RAVI index value is defined as the absolute value of the percentage difference between the 7 and 65 day averages. When a market is moving sideways, the two averages tend to have the same values, so the difference is small. Conversely, when the market is trending, the fast average rapidly pulls away from the slow average, producing a larger difference and larger index value. Generally speaking, a RAVI value below 3 percent indicates sideways prices, and above 3 percent, trending prices.

The properties parameters defined for the 65sma\_3cc strategy enable you to choose which filter you want to use (ADX or RAVI), and also let you control a variety of moving average lengths, stop loss limits, and other strategy parameter values. Chande says that this strategy (not this particular implementation of it) has been tested on 20 years of data for 23 different markets, and that it was robust and profitable on each one. So this should be a good example of a computerized trend following system.

Finally, notice that this strategy has more user-settable properties and many more helper methods than do other strategies in this document. The helper methods increase code readability because they isolate utility code into small, manageable pieces.

Here is the complete code for the strategy.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

// this enum defines allowed filter values

public enum FilterType {

None,

RAVI,

ADX

}

[StrategyComponent ("{9c5c20ff-0008-4728-9c3c-b700efbfec60}",

ComponentType.ATSComponent,

Name = "SMA\_3CC", Description = "")]

public class

**SMA\_3CC : ATSComponent {**

// the slow average is 65 bars long by default private SMA sma;

private int smaLength = 65;

// only enter new trades if no position exists private bool entryEnabled = false;

// for consecutive closes private int ccCount = 0; private int consClosesCount = 3;

// record the crossing state

private ECross smaCross = ECross.None;

// for the bar count exit method private bool barCountExitEnabled = false; private int barsToExitCount = 20; private int barsFromEntry = 0;

// for the trailing stop exit method private bool trailingStopEnabled = false; private double trailingStopLevel = 500; private ATSStop trailingStop;

// Exit when High/Low exceed previous price range private bool trendFollowingExitEnabled; private int trendFollowingExitLength = 14; private bool exitOnBarOpen = false;

// RAVI Filter parameters private SMA shortSMA; private int shortSMALength = 7; private double raviLevel = 0.5; // ADX Filter parameters private ADX adx; private int adxLength = 14; private double adxLevel = 20;

// the users choice of filter type

private FilterType filterType = FilterType.None;

// shares to buy, and trading orders private int qty = 100;

SingleOrder buyOrder;

SingleOrder sellOrder;

#region Parameters

[Category ("Parameter"), Description ("")] public int Qty { get { return qty; } set { qty = value; }

}

[OptimizationParameter (3, 30, 1)]

[Category ("Parameter"), Description ("Consecutive Closes Count")]

public int ConsClosesCount { get { return consClosesCount; } set { consClosesCount = value; }

}

[Category ("Parameter"), Description ("SMA Length")] public int SMALength { get { return smaLength; } set { smaLength = value; }

}

[Category ("Bars Exit"), Description ("Bars Exit Level")] public bool BarCountExitEnabled { get { return barCountExitEnabled; } set { barCountExitEnabled = value; }

}

[Category ("Bars Exit"), Description ("Bars to Exit Count")] public int BarsToExitCount { get { return barsToExitCount; } set { barsToExitCount = value; }

}

[Category ("Stop Exit"), Description ("Stop Exit Level")] public double TrailingStopLevel { get { return trailingStopLevel; } set { trailingStopLevel = value; }

}

[Category ("Stop Exit"), Description ("Stop Exit Enabled")] public bool TrailingStopEnabled { get { return trailingStopEnabled; } set { trailingStopEnabled = value; }

}

[Category ("Trend-Following Exit"), Description ("Trend-Following Exit Enabled")]

public bool TrendFollowingExitEnabled { get { return trendFollowingExitEnabled; } set { trendFollowingExitEnabled = value; }

}

[Category ("Trend-Following Exit"), Description ("Trend-Following Exit Length")]

public int TrendFollowingExitLength { get { return trendFollowingExitLength; } set { trendFollowingExitLength = value; }

}

[Category ("Filter"), Description ("Filter Type")] public FilterType FilterType { get { return filterType; } set { filterType = value; }

}

[Category ("ADX"), Description ("ADX Length")] public int ADXLength { get { return adxLength; } set { adxLength = value; }

}

[Category ("ADX"), Description ("ADX Level")] public double ADXLevel { get { return adxLevel; } set { adxLevel = value; }

}

[Category ("RAVI"), Description ("Short SMA Length

(RAVI)")]

public int ShortSMALength { get { return shortSMALength; } set { shortSMALength = value; }

}

[Category ("RAVI"), Description ("RAVI Percent Level")] public double RAVILevel { get { return raviLevel; } set { raviLevel = value; }

}

#endregion

public override void

**Init () {**

sma = new SMA (Bar, smaLength); sma.Color = Color.Blue;

Draw (sma, 0);

// if required, set up the RAVI moving average if (filterType == FilterType.RAVI) { shortSMA = new SMA (Bar, shortSMALength); shortSMA.Color = Color.Green;

Draw (shortSMA, 0);

}

// if required, set up the ADX moving average if (filterType == FilterType.ADX) { // ADX is a builtin function, like SMA adx = new ADX (Bar, adxLength); adx.Color = Color.Brown;

Draw (adx, 2);

}

}

public override void **OnBar (Bar bar) {**  if (sma.Count == 0) return;

// if we are using a trend-following exit and have an open

// positiong that we should close if (trendFollowingExitEnabled && HasPosition

&& Bar.Count > trendFollowingExitLength + 1) {

// check if we are long and today's close is lower than // lowest low of the last "trendFollowingExitLength" bars.

// If so, then exit on the next bar open if (Position.Side == PositionSide.Long) {

double prevLow = Bar.GetMin (Bar.Count –

(trendFollowingExitLength - 1),

Bar.Count - 2);

if (bar.Close < prevLow) exitOnBarOpen = true;

}

// check if we are short and today's close is higher than

// highest high of the last "trendFollowingExitLength" bars

// If so, exit on the next bar open if (Position.Side == PositionSide.Short) {

double prevHigh = Bar.GetMax (Bar.Count –

(trendFollowingExitLength - 1),

Bar.Count - 2);

if (bar.Close > prevHigh) exitOnBarOpen = true;

}

}

// look for N consecutive closes after a crossover

ECross cross = Bar.Crosses (sma, bar.DateTime, BarData.Close);

// if any cross occurred, reset the consecutive close count,

// and copy the cross value so we can reset our copy of it

// without wiping out the original indicator. if (cross != ECross.None) { smaCross = cross; ccCount = 0;

}

// if a cross occurred, increment the cc count, because the

// first bar counts as the first consecutive close if (smaCross != ECross.None) ccCount++;

// if we have enough consecutive closes, it's time to trade if (ccCount == consClosesCount) {

// if we have no position open, or if we have a position

// that is opposite the cross direction (ie, we need to

// close the position) if (!HasPosition

|| (Position.Side == PositionSide.Long

&& smaCross == ECross.Below)

|| (Position.Side == PositionSide.Short

&& smaCross == ECross.Above)) {

switch (filterType) {

// enter a trade if no filters are being used

case FilterType.None: { entryEnabled = true; break;

}

// enter a trade if the RAVI filter says ok case FilterType.RAVI: { entryEnabled = FilterRAVI (); break;

}

// enable a trade if the ADX filter says ok case FilterType.ADX: { entryEnabled = FilterADX (); break;

}

}

// if an entry was enabled, open a position on next bar open if (entryEnabled) exitOnBarOpen = false;

// and reset our copy of the cross status to none smaCross = ECross.None;

}

// reset the consecutive close count too ccCount = 0;

}

}

public override void **OnBarOpen (Bar bar) {**

// if we should close our position due to the trend-following exit if (exitOnBarOpen) { exitOnBarOpen = false;

// if we have a position open, close it if (HasPosition) { ClosePosition (); return;

}

}

// if we should enter a trade if (entryEnabled) { entryEnabled = false;

// and if we have no existing position if (!HasPosition) {

// go long if our bar is above the moving average if (Bar.Last.Close >= sma.Last) OpenPosition (Side.Buy);

// go short if our bar is below the moving average

if (Bar.Last.Close <= sma.Last) OpenPosition (Side.Sell);

}

// if we have an existing position, reverse it,

// because the trend direction has changed. else

ReversePosition ();

}

// else if we should be using the bar count exit instead

// of the trend following exit else {

if (barCountExitEnabled) {

// if we have a position to close,

// close the position and reset the bar counters

if (HasPosition) { barsFromEntry++;

if (barsFromEntry == barsToExitCount) { barsFromEntry = 0;

Console.WriteLine (bar.DateTime);

ClosePosition ();

}

}

// else if we have no position open,

// reset the bars count to zero for next time else

barsFromEntry = 0;

}

}

}

public override void **OnPositionChanged () {**

// every time our position size or direction changes,

// cancel the old trailing stop and set a new one CancelExit (); if (HasPosition) SetExit ();

} private void

**CancelExit () {**

// reset the bar counter and cancel the trailing stop barsFromEntry = 0; if (trailingStop != null) trailingStop.Cancel ();

}

private void **SetExit () {**

// reset the bar counter and set a new trailing stop

// Notice this stop is just an internal signal, it is

// not a real stop loss order. The real order is issued

// in OnStopExecuted, when the stop is triggered. barsFromEntry = 0; if (trailingStopEnabled)

trailingStop = SetStop (trailingStopLevel/ qty,

StopType.Trailing, StopMode.Absolute);

}

public override void **OnStopExecuted (ATSStop stop) {**

// when the stop is triggered, close the position

ClosePosition ();

} private void **ClosePosition () {**

// create and send a market order to close the position if (Position.Side == PositionSide.Long) { MarketOrder order = MarketOrder (Side.Sell, qty); order.Send ();

} else {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Send ();

}

} private void **OpenPosition (Side side) {**

// create and send a market order to open the position

MarketOrder order = MarketOrder (side, qty); if (side == Side.Buy) buyOrder = order; else sellOrder = order; order.Send ();

} private void **ReversePosition () {**

// reverse the position with a market order

// Use double the position size to flip the position if (Position.Side == PositionSide.Long) { sellOrder = MarketOrder (Side.Sell, qty \* 2); sellOrder.Send ();

} else {

buyOrder = MarketOrder (Side.Buy, qty \* 2); buyOrder.Send ();

}

} private bool

**FilterRAVI () {**

// calculate the latest RAVI value if (shortSMA.Count == 0) return false; double smaLast = sma.Last; double shortSMALast = shortSMA.Last;

double ravi = Math.Abs (smaLast - shortSMALast)

/ Math.Min (smaLast, shortSMALast)

\* 100;

// return true to accept the trade, false to block it if (ravi >= raviLevel) return true; else return false;

} private bool **FilterADX () {**  if (adx.Count == 0) return false;

// return true to accept the trade, false to block it if (adx.Last >= adxLevel) return true; else return false;

}

}

## 6.6 Gap Closing Strategies

The main idea of a gap closing strategy is that prices often reverse to close a gap that opens up between adjacent bars. “Often” means something like 60% of the time, under the right circumstances, prices will retrace to close open gaps. You can think of a price gap as people or prices overreacting to news, or overshooting the mark because they move too far, too fast. The closing of the gap can be viewed as a corrective return to more sanity. Gap closing strategies are popular with day traders and hedge funds because they are easy to recognize, easy to trade, and can be profitable if executed carefully under the right circumstances. Gap closing strategies can also be automated fairly easily.

This section describes several gap closing strategies from the book *Trade Like a Hedge Fund,* by Altucher. Altucher was a hedge fund manager, and talks about 20 successful and uncorrelated strategies in his book (not all of them can be easily automated). This section describes the rules and SmartQuant code for several of his strategies.

All of the following strategies show code from an ATSComponent that implements the strategy with an integrated framework. First the entire code for the ATSComponent is shown, and then sections of the code are reproduced with explanations.

#### 6.6.1 Stock 2% gap

This example watches for a stock to gap down 2%, then buys the gap (long) to close the position when the gap closes, or at the end of the next bar after the gap bar. So this strategy only holds a position open for 1 bar length of time. But we present the example here using daily bars, to make the explanation more intuitive.

Here is the code for the entire component, including the boilerplate sections of (1) assembly references and (2) class headers. To save space, later examples will omit the boilerplate assembly references, strategy component attributes, and class headers. These bits of code are all essentially the same, for all examples.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{9e473d29-3638-402c-8a9e-5c447608d165}",

ComponentType.ATSComponent,

Name = "Tech1\_System1\_ATS",

Description = "")]

public class

Tech1\_System1\_ATS : ATSComponent {

#region Variables

private double percent = 0.02; // 2%

// number of contracts to order private double qty = 100; private SingleOrder sellOrder; private double prevClose;

#endregion

#region Parameters

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

#endregion

#region Events public override void **OnBarOpen (Bar bar) {**

double gap\_pct = (prevClose - bar.Open) / prevClose; if (gap\_pct > percent) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech1\_System1 - Buy"; order.Send ();

}

}

public override void **OnPositionOpened () {**

sellOrder = LimitOrder (Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System1 - Sell"; sellOrder.Send ();

}

public override void **OnBar (Bar bar) {**  prevClose = bar.Close; if (HasPosition) { sellOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech1\_System1 - Sell"; order.Send ();

}

}

#endregion

}

This simple system looks for price gaps of at least 2% down, and buys long to close the gap. This example shows the use of three code events: OnBarOpen, OnPositionOpened, and OnBar.

• **Buy a stock when it gaps down more than 2% lower than the prior close.**

// gap size to watch for

private double desired\_size = 0.02; // 2%

// number of contracts to order private double qty = 100; private SingleOrder sellOrder; private double prevClose;

public override void **OnBarOpen (Bar bar) {**

// calculate the size of the current gap double gap\_pct = (prevClose - bar.Open) / prevClose; // buy at the market if the gap is one we want to play if (gap\_pct > desired\_size) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech1\_System1 - Buy"; order.Send ();

}

}

We use the OnBarOpen event to perform the opening buy action. OnBarOpen fires before each bar is calculated by the BarFactory in the market data provider. We buy on the opening of the bar (the leading edge), not on the trailing edge of the bar (which is what OnBar gives you). If the gap size is suitable, we issue a market order to buy now, before the next bar is calculated.

• **Sell at yesterday’s close if you reach it today (using a Limit order). Otherwise sell at today’s close (using a Market order).**

// install limit order at profit target when position is opened public override void **OnPositionOpened () {**

sellOrder = LimitOrder (Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System1 - Sell"; sellOrder.Send ();

}

// close position at end of day public override void **OnBar (Bar bar) {**  prevClose = bar.Close; if (HasPosition) {

// cancel the existing limit order, and issue a market order sellOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech1\_System1 - Sell"; order.Send ();

}

}

We use the OnPositionOpened event handler to automatically issue a sell limit order after the position is created for the buy order issued in OnBarOpen. The limit sell order uses the previous day close as the limit price.

We use OnBar to update the trailing prevClose variable with the current bar.Close price. Recall that OnBar fires on the trailing edge of the bar, so the current bar.Close price becomes the prevClose price for all following events. OnBar is also used to sell at today’s close if we still have a position open (maybe the limit order did not automatically fire during the construction of the current bar).

The overall sequence of events is as follows. If a suitable gap is seen between bars (remember we look for gaps in OnBarOpen), we issue an immediate market buy signal. The strategy sends the order to the broker, and when it receives confirmation, the framework opens a new position. The opening of a new position fires the OnPositionOpened event, which we catch and use to create and send a limit sell order to the broker for the buy we just made. (The broker actually passes the stop order on to the exchange servers, where the order resides until it is cancelled, expires, or is executed.) All this happens before the next bar arrives.

During the calculation of the next bar (which takes 15 minutes trading time for 15 minute bars, or all day for daily bars), the exchange is constantly comparing the limit sell order to current market prices. If the exchange prices trigger the limit order, the order is executed, the strategy is notified, and the position is closed.

When the next bar arrives, OnBar is fired, and we see the trailing edge of the bar. If we still have a position open, it means that the limit order at the exchange was never executed. So according to the strategy rules, we cancel the existing limit order and close the position at the market price.

#### 6.6.2 Stock down, stock 2% gap

This example is just like the one above, except that it also requires that the previous bar (the previous day) was a down day for the stock, in addition to gapping down 2% at today’s open.

// was the stock down yesterday? private bool downDay = false;

// gap size to watch for private double percent = 0.02;

// number of contracts to order private double qty = 100; private SingleOrder sellOrder; private double prevClose;

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

// Buy long if yesterday was a down day, and if today’s open

// price has gapped down more than 2% below yesterday’s close public override void **OnBarOpen (Bar bar) {**  if (downDay) {

double gap\_size = (prevClose – bar.Open) / prevClose;

if (gap\_size > percent) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech1\_System2 - Buy"; order.Send ();

}

}

}

// Sell at yesterday’s closing price if you reach it today (within the

// current bar period). Otherwise close position at end of trading day

// (at end of current bar).

// install a profit target limit order at yesterday’s close public override void **OnPositionOpened () {**

sellOrder = LimitOrder (Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System2 - Sell"; sellOrder.Send ();

}

// if a position exists, close it at end of trading today // The limit order was not triggered during the current bar period.

// This means the position is held open a maximum of 1 bar of time public override void **OnBar (Bar bar) {**

// update flag to remember if stock went down today downDay = false; // assume day was not down

if (bar.Open > bar.Close) // then flip if day was down

downDay = true;

// update prevClose with today’s price

prevClose = bar.Close; if (HasPosition) { sellOrder.Cancel ();

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech1\_System2 - Sell"; order.Send ();

}

}

#### 6.6.3 Stock down, stock 5% gap

This example is the same as the previous example, except that it uses 5% instead of 2% for the gap size. Increasing the gap size has the effect of reducing the number of trades (because 5% gaps are more rare than 2% gaps), but increasing the profit per trade (because the gap size is bigger).

// Buy a stock if yesterday (previous bar) was a down day (down bar),

// and if today’s (current bar) price gaps down more than 2% below // yesterday’s closing price.

// down day flag

private bool downDay = false; // watch for gaps that are 5% in size private double percent = 0.05;

// go long if yesterday was a down day, and if today’s open

// price has gapped down more than 2% public override void **OnBarOpen (Bar bar) {**  if (downDay) {

double gap\_size = (prevClose – bar.Open) / prevClose; if (gap\_size > percent) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech1\_System2 - Buy"; order.Send ();

}

}

}

#### 6.6.4 QQQQ gap down 0.5%, stock down, stock gap 5%

This example continues the trend of adding more selective conditions to the entry criteria for a trade. Now in order to enter a trade, the market must gap down 0.5% on today’s bar, the stock must be down the previous day, and the stock must gap down 5% on today’s bar at open.

This example also illustrates how you can use a “cross” ATSComponent to work with more than one instrument in your code at once. For instance, this example looks at both the Nasdaq QQQQ (hardcoded), as well as at the incoming instrument (whatever it is). In the code below, QQQQ represents the market, and the traded stock is called the “tradeInstrument.”

// gap sizes for stock (5%) and for QQQQ market (0.5%) private double percent = 5; private double symbolPercent = 0.5;

// number of contracts to order private double qty = 100;

private SingleOrder sellOrder; private double prevClose; private double prevSymbolClose; private bool downDay = false; private bool symbolGapped = false; private bool entryEnabled = false;

// the instrument being traded private Instrument tradeInstrument; // hardcode the Nasdaq 100 QQQQ ETF

private string symbol\_QQQQ = "QQQQ";

//

// User settable properties for QD property sheet

//

[Category ("Parameter"), Description ("Symbol")] public string Symbol { get { return symbol; } set { symbol = value; }

}

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

[Category ("Parameter"), Description ("SymbolPercent")] public double SymbolPercent { get { return symbolPercent; } set { symbolPercent = value; }

}

// open a trade at market open (bar open) if all the conditions are met public override void

**OnBarOpen (Instrument instrument, Bar bar) {**

// if this bar open is for the traded instrument if (instrument.Symbol != symbol\_QQQQ) {

// if yesterday was a down day, and gap is > 5%,

// enable the trade to be entered

if (downDay) {

if ((prevClose - bar.Open) / prevClose > percent / 100) { entryEnabled = true;

}

}

// remember the incoming trade instrument for later reference

if (tradeInstrument == null) tradeInstrument = instrument;

}

// else if this bar open is for the QQQQ instrument (the market) else {

// enable the trade to be entered if QQQQ gaps down enough symbolGapped = (prevSymbolClose - bar.Open) / prevSymbolClose

> symbolPercent / 100;

}

// open a long position if stock is down/gapped, and market is gapped if (entryEnabled && symbolGapped) {

MarketOrder order = MarketOrder (tradeInstrument, Side.Buy, qty); order.Text = "Tech1\_System4 - Buy"; order.Send ();

}

}

// install a limit order to take profits, if a trade position is opened public override void

**OnPositionOpened (Position position) {**  sellOrder = LimitOrder (position.Instrument, Side.Sell, qty, prevClose);

sellOrder.Text = "Tech1\_System4 - Sell"; sellOrder.Send ();

}

// on todays bar, close open positions for trade instrument

// also update variables downDay, prevClose, public override void

**OnBar (Instrument instrument, Bar bar) {**

// if bar is for trade instrument if (instrument.Symbol != symbol\_QQQQ) { // update close and downDay values prevClose = bar.Close; downDay = (bar.Open > bar.Close);

// close position if limit order was not reached if (Portfolio.Positions[instrument] != null) { sellOrder.Cancel ();

MarketOrder order = MarketOrder (instrument, Side.Sell, qty); order.Text = "Tech1\_System4 - Sell"; order.Send ();

}

}

// else if bar is for the market QQQQ else {

prevSymbolClose = bar.Close;

}

// reset position entry flags for OnBarOpen method entryEnabled = false; symbolGapped = false;

}

#### 6.6.5 QQQQ gap 0.5%, stock up, stock gap 5%

This example is the same as the previous example, except that it looks for up gaps in the market (QQQQ), a stock up day for the previous bar, and an upward stock price gap of at least 5%.

The main point of this example from the *Trade Like a Hedge Fund* book is that going short (which you do to close upward gaps) is not the exact opposite of going long (which you do to close down gaps). The book’s author argues that for some unknown reason, going short is not the exact opposite of going long—back tests of long strategies almost always outperform the “equivalent” short strategies. In other words, a strategy that works on the long side just won’t work as well on the short side. From this, it follows that you cannot match a long strategy’s performance characteristics simply by flipping positions from long to short. The author says no one knows why short strategies do not perform as well as long strategies. Perhaps it’s the natural upward bias of the markets at work. But you wouldn’t think that such a bias would have an effect over a few bars of trading time. Nonetheless, he argues, it does make a difference.

Here is the same essential code from the example above—but as you can see, code lines that referenced “downDay” code have been changed to “upDay”, and buy/sell orders have been reversed to open short and close long.

private double percent = 5; private double symbolPercent = 0.5; private double qty = 100; private SingleOrder buyOrder; private double prevClose; private double prevSymbolClose;

// use an upDay flag

private bool upDay = false; private bool symbolGapped = false; private bool entryEnabled = false; private Instrument tradeInstrument; private string symbol = "QQQQ";

[Category ("Parameter"), Description ("Symbol")] public string Symbol { get { return symbol; } set { symbol = value; }

}

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

[Category ("Parameter"), Description ("SymbolPercent")] public double SymbolPercent { get { return symbolPercent; } set { symbolPercent = value; }

}

// open a trade at market open (bar open) if all the conditions are met public override void

**OnBarOpen (Instrument instrument, Bar bar) {**  if (instrument.Symbol != symbol) { // if yesterday was an up day

if (upDay) {

if ((bar.Open - prevClose) / bar.Open > percent / 100) { entryEnabled = true;

} }

if (tradeInstrument == null) tradeInstrument = instrument;

} else {

symbolGapped = (bar.Open - prevSymbolClose) / bar.Open > (symbolPercent / 100);

}

// go SHORT to close the gap if (entryEnabled && symbolGapped) {

MarketOrder order = MarketOrder (tradeInstrument, Side.Sell, qty); order.Text = "Tech1\_System5 - Sell"; order.Send ();

}

}

// install a limit order to BUY at the profit target point public override void

**OnPositionOpened (Position position) {**

buyOrder = LimitOrder (position.Instrument, Side.Buy, qty, prevClose); buyOrder.Text = "Tech1\_System5 - Buy"; buyOrder.Send ();

}

// close the open position at market close (end of current bar)

// also update daily variables upDay, prevClose, prevSymbolClose public override void

**OnBar (Instrument instrument, Bar bar) {**  if (instrument.Symbol != symbol) { prevClose = bar.Close; upDay = (bar.Open < bar.Close);

// close open position if limit order was not reached if (Portfolio.Positions[instrument] != null) { buyOrder.Cancel ();

MarketOrder order = MarketOrder (instrument, Side.Buy, qty); order.Text = "Tech1\_System5 - Buy"; order.Send ();

} } else {

prevSymbolClose = bar.Close;

}

entryEnabled = false; symbolGapped = false;

}

#### 6.6.6 QQQQ gap 0.5%, stock down, stock gap 5%, hold overnight

This example continues the trend of examples shown previously, but removes the limit order and instead holds the position overnight (until the next bar is received).

The original gap closing strategy in the examples above always (1) opened a position at market or current bar open, (2) hoped to close the position during the trading day (during the current bar period) with a profit target limit order, and (3) closed the open position at the end of the current trading day (the end of the current bar) if the limit order was never triggered.

This strategy, instead of closing open positions at the end of the current trading day or bar, holds the positions overnight (for one more bar), thereby giving the instrument prices one more bar to close the gap before the open position is closed.

Notice that no limit order is installed when the position is first opened, since this strategy always wants to hold the position overnight (for one more bar).

private double percent = 5; private double symbolPercent = 0.5; private double qty = 100; private SingleOrder sellOrder; private double prevClose; private double prevSymbolClose; private bool downDay = false; private bool symbolGapped = false; private bool entryEnabled = false;

private Instrument tradeInstrument; private string symbol = "QQQQ";

[Category ("Parameter"), Description ("Symbol")] public string Symbol { get { return symbol; } set { symbol = value; }

}

[Category ("Parameter"), Description ("Quantity")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Percent")] public double Percent { get { return percent; } set { percent = value; }

}

[Category ("Parameter"), Description ("SymbolPercent")] public double SymbolPercent { get { return symbolPercent; } set { symbolPercent = value; }

}

// open a trade at market open (bar open) if all the conditions are met

// on market open on next day (bar), close open positions public override void

**OnBarOpen (Instrument instrument, Bar bar) {**

// if instrument is the trade instrument, not the market QQQQs if (instrument.Symbol != symbol) {

// if position has been open overnight, close it at // the market open price the next day.

if (Portfolio.Positions[instrument] != null) { if (sellOrder != null) sellOrder.Cancel ();

sellOrder = StopOrder (instrument, Side.Sell, qty, prevClose); sellOrder.Text = "Tech1\_System6 - Sell"; sellOrder.Send ();

}

// else if no position is open, try to enable a new trade // if gap size today is big enough after yesterday’s down day else { if (downDay) {

if ((prevClose - bar.Open) / prevClose > percent / 100) { entryEnabled = true;

}

} }

if (tradeInstrument == null) tradeInstrument = instrument;

}

// if bar instrument is the market QQQQs, check QQQQ gap size else {

symbolGapped = (prevSymbolClose - bar.Open) / prevSymbolClose

> symbolPercent / 100;

}

// open a new position if all conditions are met

if (entryEnabled && symbolGapped) {

MarketOrder order = MarketOrder (tradeInstrument, Side.Buy, qty); order.Text = "Tech1\_System6 - Buy"; order.Send ();

}

}

// on trailing edge of today’s bar (at market close for daily bars),

// update all variables such as downDay, prevClose, etc.

public override void

**OnBar (Instrument instrument, Bar bar) {**

// if current instrument is the trade symbol (not the QQQQs) if (instrument.Symbol != symbol) { prevClose = bar.Close; downDay = (bar.Open > bar.Close);

}

// else if current instrument is the market QQQQs else {

prevSymbolClose = bar.Close;

}

entryEnabled = false; symbolGapped = false;

}

## 6.7 Spread and Volatility Trading Strategies

The main concept behind spread trading systems is that you try to profit by trading the spread between two instruments (or two derivations from other sources of your choice). Spread trading systems are similar to gap closing systems in that when the spread (or gap) gets too big, you open a position that will profit from a closure of the spread (or gap) back to normal conditions.

To implement a spread trading strategy, one common approach is to calculate a ratio between the two price series that define the spread, and then trade the volatility of the ratio against itself (compared to its normal range of values).

For example, suppose the ratio normally has a value of 1.2 with a standard deviation of 0.2. This means that 68% of the time the spread—as indicated by the value of the ratio—would be between 1.0 and 1.4 (one standard deviation from the mean of 1.2). It also means that 94% of the time the value would be between 0.8 and 1.6 (two standard deviations from the mean of 1.2).

So if your strategy saw a ratio value of 2.0 during trading, the strategy could recognize that the current ratio value (the spread) was WAY outside of normal bounds, and could open a position to profit from a return to normal ratio conditions. A true spread trading strategy would go long on the instrument with the low price, and short on the instrument with the high price of the spread. That way, the spread strategy would not be exposed to long term price moves while the trading position was open.

The following strategy follows the trading algorithm described above—it first calculates a ratio from the two spread series, and then trades against abnormal values of the ratio. But as we shall see, this example is not a true spread trading strategy because it only trades one instrument. This makes it more of a plain volatility strategy than a spread trading strategy.

#### 6.7.1 Unilateral Pairs Trading

The concept behind this strategy is to trade the spread between the Nasdaq 100 QQQQ instrument and the S&P 500 SPY instrument. Normally these instruments travel together fairly closely, much like all three of the Dow Jones DIA, SPY, and QQQQ all travel together.

The strategy implementation calculates a ratio between QQQQ and SPY, and then trades the spread by trading deviations in the ratio from a moving average of the ratio. The SPY instrument is used as the reference instrument, since it is more stable than the QQQQ instrument. This means that when the strategy opens a position, it takes a position in the (more volatile) QQQQ only. It takes no position in the SPY instrument.

This strategy is *unilateral* because it trades only one side (QQQQ) of the two instruments in the spread. It is important to understand that this unilateral technique exposes the strategy to potentially large price movements while the position is open. This is because the strategy only has a position in one instrument (one side of the spread). For example, if the strategy is long, and both SPY and QQQQ drop together a significant amount without closing the spread, the strategy will suffer a corresponding loss.

It is worth saying that if the strategy forces positions to close within a bar or two (like many of the gap closing strategies in this document do), then the strategy exposure to long term (multiplebar) price moves against the position while it is open is probably inconsequential.

However, this particular example does not force positions to close within a bar or two, because the only exit criterion used is a return of the SPY/QQQQ ratio to within the normal standard deviation. So this particular strategy is exposed to price moves while the position is open. It hopes that the volatility of the ratio returns to normal before the instrument price and moving average can both go against the position in a significant way.

Needless to say, you should probably consider using a stop loss order on all strategies like the one shown here, just in case you enter a trade that goes against you before volatility returns to normal.

//

// Unilateral Pairs Trading Strategy from Trade Like a Hedge Fund book

// using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{7430da6e-65ec-4302-8959-706b81b832cc}",

ComponentType.ATSCrossComponent,

Name = "Tech2\_System\_CrossATS",

Description = "The Unilateral Pairs Trading System")] public class

**Tech2\_System\_CrossATS : ATSCrossComponent {**

// ratio series

// Each day, the ratio is calculated and added to this series private DoubleSeries ratioSeries;

// sma of ratio series

// Then we take a simple moving average of the daily ratios private SMA smaRatio;

// diff between ratio series and sma of ratio series

// Then we calculate the daily volatility of the ratio against the // simple moving average of itself, to see how far off normal it is.

// Each day, the diff = ratio – moving average is put into this seies private DoubleSeries diffSeries;

// sma of diff between ratio series and sma of ratio series

// Next we calculate a moving average of the differences private SMA smaDiff; // standard deviation of diff between ratio series and sma of ratio series // Finally we calculate a standard deviation for each daily diff value.

// This value says how many standard deviations the current diff value is

// from the normal mean (from the sma of the diff). If this number gets

// to be too big, then the strategy enters a trade. private SMD smdDiff;

// Number of contracts to order

private double qty = 100;

// Upper and lower bounds of Standard Deviation

// Strategy opens a position when current ratio value is more

// than 1.5 standard deviations from the mean. private double entryUpperStdDev = 1.5; private double entryLowerStdDev = -1.5;

// Strategy closes a position when ratio value returns to

// being within 0.5 standard deviations from the mean private double exitUpperStdDev = 0.5; private double exitLowerStdDev = -0.5;

// Upper and lower bounds of Percent private double upPercent = 2; private double downPercent = 2; private double mainLast = -1; private double secondLast = -1; private string mainSymbol = "QQQQ"; private string secondSymbol = "SPY";

private Instrument mainInstrument; // reference the QQQQs private Instrument secondInstrument; // trade the SPYs private double mainPrev;

[Category ("Symbols"), Description ("Main (traded) Stock' Symbol")] public string MainSymbol { get { return mainSymbol; } set { mainSymbol = value; }

}

[Category ("Symbols"), Description ("Second (pair for the main) Stock' Symbol")]

public string SecondSymbol { get { return secondSymbol; } set { secondSymbol = value; }

}

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Entry Upper bound of Standard Deviation")]

public double EntryUpperStdDev { get { return entryUpperStdDev; } set { entryUpperStdDev = value; }

}

[Category ("Parameter"), Description ("Entry Lower bound of Standard Deviation")]

public double EntryLowerStdDev { get { return entryLowerStdDev; } set { entryLowerStdDev = value; }

}

[Category ("Parameter"), Description ("Exit Upper bound of Standard Deviation")]

public double ExitUpperStdDev { get { return exitUpperStdDev; } set { exitUpperStdDev = value; }

}

[Category ("Parameter"), Description ("Exit Lower bound of Standard Deviation")]

public double ExitLowerStdDev { get { return exitLowerStdDev; } set { exitLowerStdDev = value; }

}

// The initialization routine creates various objects and attaches

// them to the class variables. It also specifies what lines to

// draw on the QD bar chart.

public override void

**Init () {**

// create series objects and moving average objects ratioSeries = new DoubleSeries (); diffSeries = new DoubleSeries (); smaRatio = new SMA (ratioSeries, 20); smaDiff = new SMA (diffSeries, 20); smdDiff = new SMD (diffSeries, 20);

// get references to SPY and QQQQ from instrument manager mainInstrument = InstrumentManager.Instruments[mainSymbol]; secondInstrument = InstrumentManager.Instruments[secondSymbol];

// specify drawing colors and barchart pad locations smaRatio.Color = Color.Red; ratioSeries.Color = Color.Red; smaRatio.Color = Color.Red; smdDiff.Color = Color.Red;

Draw (mainInstrument, smaRatio, 4);

Draw (mainInstrument, ratioSeries, 5);

Draw (mainInstrument, smaRatio, 5);

Draw (mainInstrument, smdDiff, 6);

}

The following OnBarSlice event handler is called after a BarFactory has finished emitting bars for all instruments used by the strategy. You should use the OnBarSlice event when your strategy wants to compare or reference multiple instruments within the same invocation of an event handler. In this particular strategy, we need to reference both the SPY and QQQQ instruments in the same code invocation in order to calculate the ratio between them.

public override void **OnBarSlice (long barSize) {**

DateTime date = Clock.Now;

// if we have ratio values to work with

if (mainLast != -1) {

// calculate today’s ratio and add the ratio to the ratio series double ratio = mainLast / secondLast; ratioSeries.Add (date, ratio);

// if there are values in the moving average series,

// calculate diff between today’s ratio and its moving average, // then add the difference value to the diff series if (smaRatio.Count > 0)

diffSeries.Add (date, ratio - smaRatio[date]);

}

// if we have no moving average of ratio-ratioSMA diff values,

// we can’t do anything yet, so return

if (smaDiff.Count == 0) return;

// if we reach this point, we have a moving average of ratio-ratioSMA // to work with. This moving average lets us calculate std deviations.

// compare todays diff (diffseries) with the SMA of diffs (smaDiff) // and divide by the standard deviation (smdDiff) denominator // Now we can tell if the current ratio is way off normal.

double stdDev = (diffSeries[date] - smaDiff[date]) / smdDiff[date];

// if we have no existing position, see if you can enter a trade if (Portfolio.Positions[mainInstrument] == null) {

// if current ratio-smaDiff is more than 1.5 standard deviations // from the mean, open a position (either long or short)

if (stdDev > entryUpperStdDev

&& (mainLast - mainPrev) / mainPrev >= upPercent / 100)

SendOrder (Side.SellShort); if (stdDev < entryLowerStdDev

&& (mainLast - mainPrev) / mainPrev <= -downPercent / 100)

SendOrder (Side.Buy);

}

// else if we already have an open position, try to close it if

// current ratio-smaDiff value has returned inside 0.5 standard

// deviations from the mean

else {

if (Portfolio.Positions[mainInstrument].Side ==

PositionSide.Short) {

// if you are short, buy to close the position

if (stdDev < exitUpperStdDev)

SendOrder (Side.Buy);

} else {

if (stdDev > exitLowerStdDev) SendOrder (Side.Sell);

}

}

}

// In this strategy, most of the action takes place in the

// OnBarSlice method, which acts like OnBar but for multiple

// instruments. So the following OnBar method, which can only

// “see” one instrument at a time, is only used to update daily // variables. Remember that OnBarSlice can “see” all instruments.

//

public override void

**OnBar (Instrument instrument, Bar bar) {**  if (instrument == mainInstrument) { mainPrev = mainLast; // yesterday’s SPY closing value mainLast = bar.Close; // today’s SPY closing value

} else

secondLast = bar.Close;

}

// this is a helper method to open or close a position, using

// a market order private void

**SendOrder (Side side) {**

MarketOrder order = MarketOrder (mainInstrument, side, qty);

order.Text = "Tech2\_System " + side.ToString () + " " +

mainInstrument.Symbol; order.Send ();

}

}

#### 6.7.2 QQQ Crash, QQQQ Trade

The Nasdaq 100 (QQQ) index is fairly volatile because it contains so many technology stocks whose prices move around so much more than average non-technology stocks. The volatility of the QQQ has many sources, including rapid technological advances, competition, and sudden news that can affect stock prices.

The strategy below waits for a big “crash” in the QQQ, where a crash is defined as QQQ prices moving below a lower Bollinger band set 1.5 standard deviations away from a 10-day moving average. While using 1.5 standard deviations is not unusual for Bollinger bands, the use of 1.5 with a short 10-day moving average is an unusual combination.

To move below a lower Bollinger band that is based on a 10 day average requires a pretty drastic price move, because a fast 10-day average will stick closely to price movements. So this strategy looks for drastic drops in QQQ, buys long the morning after the crash (in OnBarOpen), and then trades the gap back up as usual. It exits when a profit is made, or after 30 days.

We put this strategy under the volatility section because it uses methods that deliberately select volatile moves (1.5 standard deviations, 10 day average). In contrast, our other Bollinger band examples are listed under the range trading section, because they focus on slower moving averages and less volatile price moves.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{b743a27b-a8ad-4abd-9568-d15edb26c733}",

ComponentType.ATSComponent, Name = "Tech8\_System1\_ATS",

Description = "The QQQQ Crash System")] public class

**Tech8\_System1\_ATS : ATSComponent {**

// bollinger band, 10 days, 1.5 std deviation private BBL bbl; private int bblLength = 10;

private double bblOrder = 1.5;

// remember what you paid on entry private double entryPrice;

// exit after 30 days int maxDuration = 30;

// orders and quantity

SingleOrder buyOrder; private double qty = 100;

[Category ("Parameter"), Description ("Max number of days, while position is active")] public int MaxDuration { get { return maxDuration; } set { maxDuration = value; }

}

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Length of BBL")] public int BBLLength { get { return bblLength; } set { bblLength = value; }

}

[Category ("Parameter"), Description ("Order of BBL")] public double BBLOrder { get { return bblOrder; } set { bblOrder = value; }

}

public override void

**Init () {**

// get a reference to bar series for current instrument

BarSeries bars = Bars[Instrument]; // set up bollinger bands

bbl = new BBL (bars, bblLength, bblOrder);

bbl.Color = Color.Blue;

Draw (bbl, 0);

}

public override void **OnBar (Bar bar) {**

// if we don't have a position and we have some bars // in the bollinger series, try to enter a new trade if (!HasPosition) { if (bbl.Count > 0) {

// if the current bar is below the lower bollinger band

// buy long to close the gap

if (Bar.Crosses (bbl, bar.DateTime) == ECross.Below) { buyOrder = MarketOrder (Side.Buy, qty); buyOrder.Text = "Tech8\_System - Buy"; buyOrder.Send ();

}

}

}

// else if we DO have an existing position, and if

// today's bar is above our entry price (profitable),

// then close the position with a market order else {

if (entryPrice < bar.Close) {

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech8\_System - Sell"; order.Send ();

}

}

}

public override void **OnPositionOpened () {**

// when a position is opened, remember what we paid for

// the instrument. Notice we use the average price of

// all the partial fills that we might have received. entryPrice = buyOrder.AvgPx; // when a position is opened, set a stop to remind us // to exit the position once 30 days have elapsed.

SetStop (Clock.Now.AddDays (maxDuration));

}

public override void

**OnStopExecuted (ATSStop stop) {**

// if our 30-day stop reminder is executed, then // close the existing position with a market order.

// This order action should probably be protected with a

// "if (HasPosition)" clause, to avoid issuing an order

// when we really don't have a position to close MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech8\_System - Sell"; order.Send ();

}

}

#### 6.7.3 QQQ Crash, Stock Trade

This example is just like the one above, except that instead of trading the Nasdaq 100 QQQQ ETF (Exchange Traded Fund), it trades a more volatile high-beta stock component of the QQQ index. The main idea here is that when the market (QQQ) falls fast, buy high beta stocks to play the bounce and close the gap upwards.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{0c67a3b7-d918-4e7f-8ef5-dec6654f6183}",

ComponentType.ATSCrossComponent, Name = "Tech8\_System2\_CrossATS",

Description = "QQQQ Crash Stocks")] public class

**Tech8\_System2\_CrossATS : ATSCrossComponent {**

// for bollinger bands private BBL bbl; private int bblLength = 10; private double bblOrder = 1.5;

// remember what you paid on entry private double entryPrice;

// exit after 30 days int maxDuration = 30;

// orders and quantity

SingleOrder buyOrder; private double qty = 100;

// QQQQ represents the market private string marketSymbol = "QQQQ"; Instrument marketInstrument; // this is the stock we buy and sell private string tradeSymbol = "AMZN";

Instrument tradeInstrument;

[Category ("Symbols"), Description ("Main (traded) Stock' Symbol")] public string MainSymbol { get { return marketSymbol; } set { marketSymbol = value; }

}

[Category ("Symbols"), Description ("Second (pair for the main) Stock' Symbol")]

public string SecondSymbol { get { return tradeSymbol; } set { tradeSymbol = value; }

}

[Category ("Parameter"), Description ("Max number of days, while position is active")] public int MaxDuration { get { return maxDuration; } set { maxDuration = value; }

}

[Category ("Parameter"), Description ("Number of contracts to order")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Length of BBL")] public int BBLLength { get { return bblLength; } set { bblLength = value; }

}

[Category ("Parameter"), Description ("Order of BBL")] public double BBLOrder { get { return bblOrder; } set { bblOrder = value; }

}

public override void

**Init () {**

// get references to instruments

marketInstrument = InstrumentManager.Instruments[marketSymbol]; tradeInstrument = InstrumentManager.Instruments[tradeSymbol];

BarSeries bars = Bars[marketInstrument]; // set up bollinger band series bbl = new BBL (bars, bblLength, bblOrder); bbl.Color = Color.Blue; Draw (marketInstrument, bbl, 0);

}

public override void

**OnBar (Instrument instrument, Bar bar) {**

// if the current instrument is the market bar, and if

// we have no position in the trade instrument, then

// maybe enter a trade if criteria are met if (instrument == marketInstrument) {

if (Portfolio.Positions[tradeSymbol] == null) {

// if there are enough bars in the bollinger series if (bbl.Count > 0) {

// if current market bar is below the lower bollinger band

// then open a long position to trade the gap

if (Bars[marketInstrument].Crosses (bbl, bar.DateTime) ==

ECross.Below) {

buyOrder = MarketOrder

(tradeInstrument, Side.Buy, qty);

buyOrder.Text = "Tech8\_System2 - Buy"; buyOrder.Send ();

}

}

}

}

// if the current bar is the trade instrument, and we have

// a position open, then test for a profit target exit if (instrument == tradeInstrument) {

if (Portfolio.Positions[tradeSymbol] != null) {

// if position is profitable, then close it with

// a market order. Hopefully the market order slippage

// will not wipe out your profit! if (entryPrice < bar.Close) {

MarketOrder order = MarketOrder

(tradeInstrument, Side.Sell, qty);

order.Text = "Tech8\_System2 - Sell"; order.Send ();

}

}

}

}

public override void

**OnPositionOpened (Position position) {**

// when a position is opened, remember what we paid for

// the instrument. Notice we use the average price of

// all the partial fills that we might have received. entryPrice = buyOrder.AvgPx;

// when a position is opened, set a stop to remind us

// to exit the position once 30 days have elapsed.

SetStop (position, Clock.Now.AddDays (maxDuration));

}

public override void

**OnStopExecuted (ATSStop stop) {**

// when our stop fires, close position with market order // this action should be inside a "if (HasPostion)" test MarketOrder order = MarketOrder (tradeInstrument, Side.Sell, qty); order.Text = "Tech8\_System2 - Sell"; order.Send ();

}

}

#### 6.7.4 Bollinger Bands with 5-Minute Bars

This Bollinger band strategy is intended for short intraday trades that last no longer than 5 or 10 minutes. It enters a trade if prices fall 3% below the lower Bollinger band, and exits a trade when a 1% profit target has been reached, or when two bars have passed since the trade was entered.

There are several interesting features of this system. It uses flags to say when a trade can be entered or exited (when two bars have passed). It uses the OnTrade event handler to look at trade data during bar formation intervals (so the strategy is working within two timeframes—trades and bars). And it uses SetStop method to set a stop reminder based on time (5 minutes) to remind the strategy to close the position automatically after 5 minutes have passed with an open position.

using System; using System.Drawing; using System.ComponentModel;

using SmartQuant; using SmartQuant.FIX; using SmartQuant.Data; using SmartQuant.FIXData; using SmartQuant.Trading; using SmartQuant.Series; using SmartQuant.Optimization; using SmartQuant.Indicators; using SmartQuant.Instruments; using SmartQuant.Execution;

[StrategyComponent ("{a198d354-5be3-4b00-bd3b-b62e9bb1652f}",

ComponentType.ATSComponent,

Name = "Tech16\_System1\_ATS",

Description = "Five-Minute Bollinger Band System")] public class

**Tech16\_System1\_ATS : ATSComponent {**

// bollinger bands, 10 day, 2 standard deviations (94%) private BBL bbl; private int bblLength = 10; private double bblOrder = 2;

// exit on a profit target of 1% private double qty = 1;

// enter if stock drops 3 entry\_percent below lower band private double entry\_percent = 3;

// hold it for 5 minutes minimum (to end of next 5-min bar) private int minutesToHold = 5;

// exit on the second bar after stock was bought private bool exitOnBar = false; private bool entryEnabled = true; private double prevPrice = -1; private double prevBBL = -1;

[Category ("Parameter"), Description ("Percent")] public double Percent {

get { return entry\_percent; } set { entry\_percent = value; }

}

[Category ("Parameter"), Description ("# of minutes to hold")] public int MinutesToHold { get { return minutesToHold; } set { minutesToHold = value; }

}

[Category ("Parameter"), Description ("Qty")] public double Qty { get { return qty; } set { qty = value; }

}

[Category ("Parameter"), Description ("Length of BBL")] public int BBLLength { get { return bblLength; } set { bblLength = value; }

}

[Category ("Parameter"), Description ("Order of BBL")] public double BBLOrder { get { return bblOrder; } set { bblOrder = value; }

}

public override void

**Init () {**

// get reference to bar series for current instrument

BarSeries bars = Bars[Instrument]; // set up bollinger bands bbl = new BBL (bars, bblLength, bblOrder);

bbl.Color = Color.Blue; Draw (bbl, 0);

// initialize temp vars prevBBL = -1; prevPrice = -1;

}

public override void **OnBar (Bar bar) {**

// update previous bb limit if (bbl.Contains (bar.DateTime)) prevBBL = bbl[bar.DateTime];

// if we are supposed to exit on this bar, reset flag

// and close position with market order if (HasPosition && exitOnBar) {

// close position with market order

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech16\_System1 - Sell"; order.Send ();

// enable another trading cycle exitOnBar = false; entryEnabled = true;

}

}

public override void **OnTrade (Trade trade) {**

// if we have recorded a previous bbl value, and a

// previous trade price value, then do the calculation

// (We need these two values for the calculation) if (prevBBL != -1) { if (prevPrice != -1) {

// if we do not have a position, and we are allowed

// to open a position, see if the stock price is

// 3 percent below the lower bbl band. if (!HasPosition && entryEnabled) {

// calculate the gap down size, in entry\_percent double bbl\_price\_diff = prevBBL - trade.Price; double drop\_percent = (bbl\_price\_diff / prevBBL) \* 100;

// if the stock has dropped low enough open

// a long position with a market order if (drop\_percent > entry\_percent) {

MarketOrder order = MarketOrder (Side.Buy, qty); order.Text = "Tech16\_System1 - Buy"; order.Send ();

// disable entries while we are in a position entryEnabled = false;

}

}

}

// remember the previous trade price (not bar price) prevPrice = trade.Price;

}

}

public override void **OnPositionValueChanged () {**

// this event fires every time a new trade price arrives

// and changes the market value of our position. So this

// method could execute many, many times before the next // bar arrives in the strategy.

// if we have a position, see if it is 1% profitable yet.

// If so, close the position with a market order if (HasPosition) {

if (Position.GetPnLPercent () > entry\_percent) {

MarketOrder order = MarketOrder (Side.Sell, qty); order.Text = "Tech16\_System1 - Sell"; order.Send ();

// reset control flags for another trade cycle exitOnBar = false; entryEnabled = true;

}

}

}

public override void **OnPositionOpened () {**

// when a new position is opened, set a stop reminder

// for 5 minutes in the future. After five minutes // have passed, we close the position on the next bar.

SetStop (Clock.Now.AddMinutes (minutesToHold));

}

public override void

**OnStopExecuted (ATSStop stop) {**

// When our 5 minute hold period expires, set a flag

// to exit the position on the next bar exitOnBar = true;

}

}

# Bibliography

*Beyond Technical Analysis 2ed* by Tushar S Chande. This is a superb book on developing and implementing trading systems (mechanical or manual). If you intend to spend any significant time or money on developing a trading system, this is a book that you should read first.

*Trade like a Hedge Fund* by James Altucher. This book provides “20 successful uncorrelated strategies and techniques to winning profits” (not all of which can be automated). Many of the strategies in this book are used as examples later in this document.

# 7 Terminology

Inconsistent terminology makes it harder for readers to understand new concepts of any kind. To help avoid that problem in this document, we provide the following consistent definitions for words that are used throughout SmartQuant documentation. We do not claim that the definitions below are universally accepted in the industry—but they are good enough and consistent enough to meet the needs of the discussions in this document.

**Component.** A component is a computer file that contains one or more software components. For example, a component could be an Entry or Exit component that is listed in the Components panel of the IDE. Component is a technical term that means “a computer file,” not a functional strategy term that means a “functional component” such as Entry or Exit.

**Partitioned Framework.** A partitioned framework *partitions* (splits) the functions of a strategy (such as Entry and Exit) into *multiple* components (files). Using separate components makes it easier for developers to mix and match individual strategy functions into an overall strategy that they like (simply by right clicking and choosing the components of interest). When you create a new IDE strategy project, you must choose a partitioned or an integrated strategy framework.

**Integrated Framework.** An integrated framework *integrates* the functions of a strategy (such as Entry, Exit, Market Manager, and Risk Manager) into a *single* component. An integrated strategy framework makes it easier for developers to share trading information—such as instruments, prices, and trades—among the functions of a strategy. (In contrast, information is difficult to share across component boundaries in the partitioned framework). When you create a new IDE project, you must choose a partitioned or an integrated strategy framework.

**Unaggregated Market Data.** Unaggregated market data originates with a trading exchange, passes through a market data provider (or the database equivalent of a provider, in simulation mode), and is received by your strategy without aggregation processing. That is, no aggregation operations are performed to group, simplify, or otherwise change the original data flow. The term unaggregated data typically refers to Quotes and Trades.

**Quote Data.** Quote data originates with a market data provider, and contains at least a bid/ask pair of prices and a timestamp. (SmartQuant quotes contain the best bid/ask quotes among active providers.) Quotes may contain other information as well. In a SmartQuant strategy component, incoming quotes are usually handled by an OnQuote event handler method.

**Trade Data.** Trade data originates with a market data provider, and contains at least an instrument name, trade price, timestamp, and volume (size) of the trade. In a SmartQuant strategy component, incoming trades are usually processed by an OnTrade event hander method. Trade data is also known as “Time and Sales” data.

**Time and Sales Data:** Time and sales data is another industry term for trade data. This definition is provided here for completeness—the term “trade and sales” is not used in this document.

**TAQ Data.** TAQ is short for “trades and quotes,” and normally refers to the NYSE trade and quote database. The term TAQ is listed here for completeness—it is not used in this document.

**Tick.** A tick is the minimum price move allowed for a particular instrument. Tick sizes are different for different financial instruments. For example, a tick might be 0.01 on a stock price, 0.05 on a futures contract, or 0.10 on an index option. Ticks are properties or attributes of a financial instrument, and are defined by the exchanges that trade instruments. At least in this document, ticks have nothing to do with quotes or trades. (Although some people use the term “ticks” to refer generally to trade and quote TAQ data.)

**Tick Data.** Tick data is a general term used to refer to *unaggregated data*. That is, *tick data* can refer to quote data, trade data, market depth data, or any other kind of unaggregated data.

**Aggregated Data.** Aggregated data is formed by processing *tick data* (unaggregated quote and trade data) into structured data objects that are more useful for your strategies. For example, Bar data is the most obvious type of aggregated data.

**Bar Data.** Bar data originates in the SmartQuant BarFactory, and is constructed from unaggregated quote and trade data. To see the BarFactory for a market data provider, select a provider in the Providers panel, and view its properties in the Properties panel.

**Instrument.** A financial *instrument* is something that carries financial value, such as a stock, a bond, a futures contract, or an equity option.

**Technical Indicator.** A *technical indicator* is a mathematical, formula-calculated value that indicates some interesting property of the underlying financial instrument. For example, one technical indicator is a simple 10-day moving average of daily closing price.