AUTOMATED TRADING SYSTEM

by

Duc Minh LE
MSc Computational Finance

2009

Acknowledgement

I wish to express my gratitude to my supervisor, Professor Roland Backhouse for his advice and guidance during the course of the dissertation.

I would also like to thank my family and my friends for being supportive and encouraging during difficult times.

Abstract

This is a final report which is part of the dissertation for the Master course Computation Finance. The title of this dissertation is “Automated Trading System”. As the name explains itself, this project is about building an automated trading system which employs the statistical arbitrage trading algorithm. In general, the project contains a mixture of computer science and quantitative finance. For the purpose of simplicity, the system is designed with a simple user interface and streamlined business logic compare to a real world commercial trading system. The purpose of building this system is to build a starting point so that having a profitable trading algorithm implemented into the system will increase the probability of having a profitable trading system. The build phase of the project utilises Visual C++ programming language within the .NET framework.

Keywords

Automated Trading System, Quantitative Analysis, Statistical Arbitrage, Visual C++, Visual Studio, .NET, Producer/Consumer Design Pattern, K|V Trading System Development Method, Trading Strategies.

Table of Contents

List of Figures

Chapter 1

Chapter 3

Chapter 4

Chapter 1

Introduction

In the last two decades, the fast evolvement of the financial market worldwide has led to the introduction of many complex instruments together with complex trading strategies to deal with the fast and constantly changing market conditions. In this market, a lot of profitable opportunities come and go in seconds which proves impossible for human to react to those opportunities. Therefore, developers and quantitative finance experts have come together to develop automated trading systems which captures the fast movement of market prices. Coupled with pre-defined trading strategies, the system can react to those movements quickly which offers the opportunity to make potential profits. Hence, a robust and reliable trading system and a fully tested trading strategy play a vital role in minimizing the risk and maximizing returns.

1 1.1 Automated Trading System

Basically, an automated trading system is a system which is connected to financial exchanges and is able to electronically submit trades to the exchange. Basic characteristics of a trade which includes the bid (desirable price at which to buy), or ask (desirable price at which to sell), volume (amount of the asset traded, or number of shares of it is equity trading), and when to enter the trades to the exchange, are decided based on pre-defined trading strategies. Having said that, a fully tested profitable trade strategy and a reliable trading system can help investors or traders maximize their returns and minimize their risks. One of the advantages that stands out is that the automated trading system enters trades with no emotions involved in the process so consistent results are obtained for the trader.

Automated trading systems are created by converting trading strategies into sets of machine codes. The application then runs those rules and looks for trades that adhere to the set of strategies. Apart from some benefits stated above, automated trading system also has some downsides. If the system is not properly implemented and tested, large losses can incur very quickly. Furthermore, in real life there are trading strategies that can not be transformed into algorithms (possible reasons are that the strategy requires characteristics which are beyond the accommodation by a computational system) which make it difficult for developers to fully implement the strategies.

The methodology that involves research and building trading models based purely on market price movements is called technical analysis. The concepts behind technical analysis strongly support the idea of an automated trading system. Another well-known methodology is called fundamental analysis which partly involves technical analysis and other factors such as economic, financial or qualitative factors.

2 1.2 Motivation

The motivation of this project is to research current architectures of automated trading system and implement a simple working automated trading system which can act as a backbone for future work and research in the field of developing trading systems. Besides after the

completion, the project can serve as a starting point for other developers who wish to enter the field of automated trading system development so that with further research they will be able to build more complex and more interactive systems. Furthermore, once there is a complete system running, it would be easier for developer to implement profitable trading strategies.

With the ability to emulate the movements of market prices and electronic trade submissions, this system can also be extended to back-test trading models and strategies. (Back-test: testing on the profit and loss of a trading strategy on historical prices). Hence this project can also be regarded as a tool to carry out quantitative trading models testing.

3 1.3 Objectives

Following is the set of objectives defined for this project:

- Building a automated trading system implemented in Visual C++ .NET.
- As the focus of the project is more towards the technical side, a simple trading strategy is employed for the purpose of testing the functionalities of the system. After looking at available strategies, it is decided that the statistical arbitrage using z-transform is used (the concepts will be discussed in more details in the following chapters).
- The system testing is carried out using market data generated by a market simulator (the simulator is wrapped inside a Dynamic Linked Library (DLL) TraderAPI.dll).
- Implementation phase needs to obtain objectives including a refined system which delivers good user interface, code readability, reusability and importantly system scalability for future improvements and expansions.

  Therefore this project will require research into different architectures of automated trading system and learning visual C++ using .NET framework.

4 1.4 Methodology

Similar to a normal software development cycle, there are several steps that are involved in creating an automated trading system. Van Vliet and Dr Andrew Kumiega have created a methodology which is dedicated to developing trading system in general. The steps within the methodology are based on a standard software development cycle (such as the waterfall). It has four steps: research and documentation of trading strategies, back testing, build the system and risk management. The scope of this project will focus on the third stage which is the actual build of the system. The trading strategy in this project is employed within the phase of system testing (test the success of system implementation), hence research of trading strategies to develop a profitable model will not fall into the project's scope. Having said that, back-testing the trading model will also be limited to the trading strategy that is implemented in the system. This also applies to risk management (including profit and loss reports) which will not a significant role. More details of the methodology, the K|V trading system development methodology, can be found in Van Vliet, 2007.

The methodology combines a new product development process with the well-known waterfall and spiral software development methodologies into a single paradigm for trading system development. While the four stages progress in a traditional waterfall, at the completion of each stage is a gate, at each gate a decision must be made to whether it is appropriate to move on

the next stage. The following are the four stages and their respective components (Van Vliet, 2007):

- Stage 1: Research and document trading strategies
    1. Describe Trading Idea
    2. Research Quantitative Methods
    3. Prototype in Modelling Software
    4. Check Performance
- Stage 2: Back Test
    1. Gather historical data
    2. Develop cleaning algorithms
    3. Perform in sample/out of sample tests
    4. Shadow trade and performance testing
- Stage 3: Build
    1. Build software requirement specification documents
    2. Design system architecture
    3. Program and document the system
    4. Probationary trade and performance testing
- Stage 4: Risk management
    1. Monitor portfolio statistics
    2. Perform risk calculations
    3. Document profit and loss attribution
    4. Determine causes of variation in performance

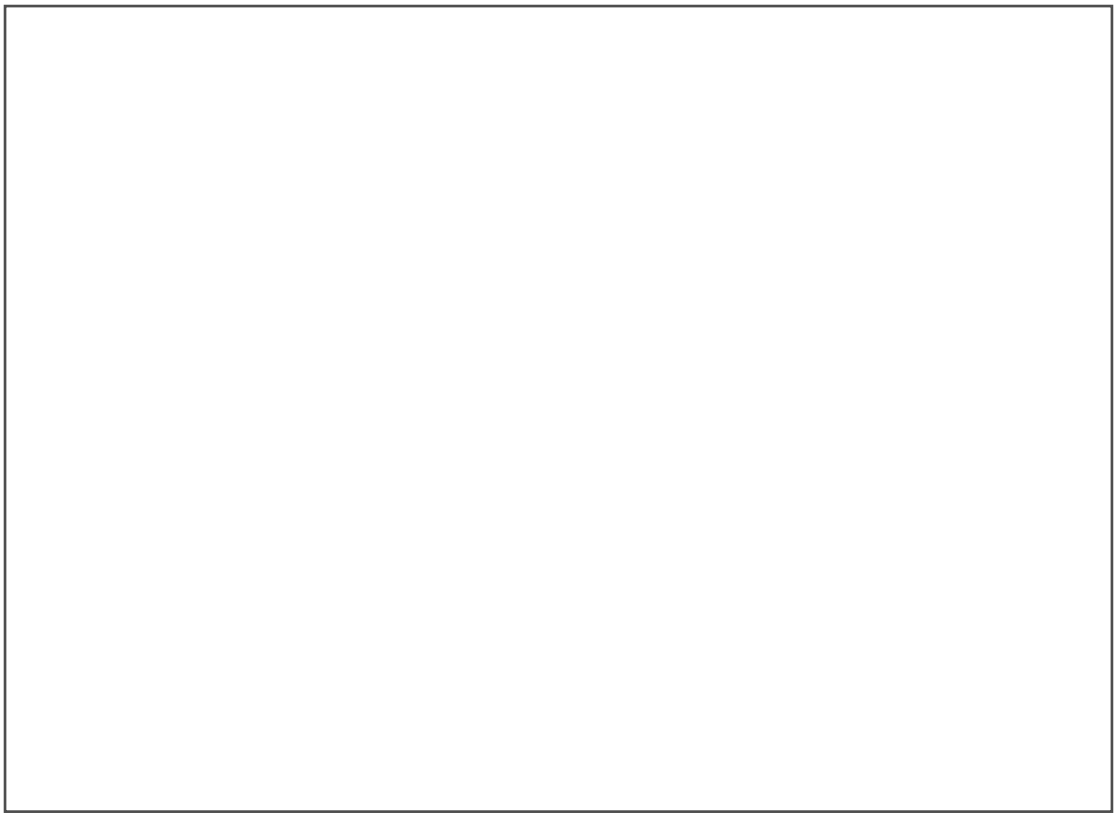

Figure 1.1: K|V trading system development methodology (Van Vliet, 2007)

1 Research and document trading strategies

The first part of this step is to be able to clearly state the trading ideas behind the system. This will serve as the foundation to the whole project. It is a fact in the real world that traders or investors often take the research and development trading models stage not significantly enough which results in unsustainable returns and often not scalable in terms of adapting to fast changing markets. Therefore, having a thorough research into trading strategies will have a big impact in generating a logical and robust trading system which eventually could offer rewarding and sustainable returns.

Building on those ideas, research needs to be carried out on existing mathematical models. The goal of this process is to refine the trading ideas and come up with the most suitable trading strategy. Furthermore the research may expand into gaining an understanding of the methodologies of other successful and well-known systems. The ultimate goal of this stage is to produce an optimal trading strategy, or maximizing returns while minimizing risk and transaction costs.

Next step, after having the right model, is to build a prototype for the model. The purpose of this step is to have a foundation so that when trading strategies change to accommodate changing market conditions, it would become easy to make modification to the prototype. Furthermore, if back testing the trading strategy leads to a conclusion that we have to back to refining the strategy, generations can be built upon the prototype foundation.

The first performance check on the strategy may not be performed thoroughly but it can produce a benchmark upon which future performance measure can be based.

2 Back Test

In order to be able to back test the performance we need to gather historical data. The larger the amount of data for testing, the more thorough the test becomes and hence a true performance of the strategy can be assessed. Having said that, we need to have a clear limit on how far back the historical data set should be as in a fast changing investment world sets of historical data in the last, say, 10 years seems unreasonable in terms of balance between how good the trading strategy performs on historical data and the cost of obtaining those data sets.

The next step is to develop cleaning algorithms. This step has nothing to do with the trading strategy research. In fact they are algorithms used to detect and remove “dirty” data from the set of historical data. In the trading world dirty data often is incorrect, contains errors which can become misleading for developers to appropriately back-test the trading strategy.

Performing a proper in sample and out of sample tests is one of the most critical steps in the process. An out of sample set contains data that does not conform to the mean of the sample. Opposite of out of sample data set is in sample data set. Trading algorithms must be tested against both these tests in order to progress to the next stage. A well-developed algorithm will perform consistently in both tests. Especially, if the trading strategy is tested to perform well enough with the out of sample data set, the traders can be confident that the system can cope well when markets conditions are extreme (for example, when economy is in a depression state). There are three possible outcomes that can come out from the tests: profitable in both in sample and out of sample tests, profitable in sample but not out of sample and unprofitable in both tests. Apparently every automated trading system aims for the first outcome.

At the point of carrying out shadow trades, developers often use the prototype. The purpose of the trades is only for understanding the performance management tools, therefore the profit and loss of the shadow trade is not indicative of the profit and loss of the system when completed. Probationary trading may occur for a limited period of time and with a small amount of capital in order to more fully understand the behaviour of the system and to understand what tools will be needed by management. Probationary trading is the process of testing the trading strategy in a trading system, the purpose of this type of trading is to test both the trading model and the trading system. Probationary trading is often performed using shadow trades which mean that trading is going on a simulated world.

3 Build

Similar to other software development projects, having a software requirement specification which defines in detail all the functionalities of the system is vital before jumping into the actual technical implementation of the system. This type of document will allow the system developers to quickly build the system with the correct functionalities and to the proper specifications. Furthermore it also includes the steps for implementing the system along with documentation on trading models, GUI requirements, reports generation tools.

The next step to take is based on the functionalities required for the system, a document of system architecture and design is drawn up. They should include financial calculations, real-time data and user interfaces, order routing connections, reporting functionalities, and any other necessary processes.

Once the hardware is built and network connections are completed, the process of construction will be a step-by-step march through the Software Architecture Document. Since the

Software Architecture Document will be evolving, the ideal solution is to continue to add and refine it as the system is built. It should be noted that programming the system also includes proper software testing.

Once the coding of the system is complete and the system is up and running without any run-time errors, another probationary trade should be performed on the system. This time real data from markets should be used so that the true performance of the system can be assessed.

Before moving to the next stage, based on the performance from the probationary trade, there should be a few iterations of this stage to refine and improve the system in overall. Moreover, this is a chance for the developers to spot any design flaws in trading algorithm or trading system. Another purpose of this stage is to allow the investor to determine, based on the current performance of the model, whether he/she is happy with the balance of profit and loss and risk returned by the trading strategy.

4 Risk Management

A trading world involves fast moving markets which to maintain consistent returns, constant monitoring of the performance of the trading system is required. Hence the trading success necessitates the requirement of periodic reports. Therefore, a system for monitoring and reporting profit and loss, historical trades, and risk factors must be implemented. Essentially, these reports will help us understand whether the system is working within specifications and the parameters of the back test.

Risk calculation reports will help us understand whether the system is taking too much risk. They also help in explaining the relative of the system to the market overall performance. Risk in the scope of this project should be understood as how much of a loss the trading strategy makes in average when the price moves in an unfavourable direction. Based on the amount of risk the trading system is exposed to, the investor can also determine the resistance level (at which the trading model goes “bust”)s.

A good way to monitor the success of a system is to keep track of individual trades and their respective payoffs. These will be valuable when re-evaluating the underlying premise for the system relative to a benchmark.

Finally, understanding why there are variations in the system performance is key to finding out possible logic leaks in the trading strategy. If the cause to the variations is found, a fix can be applied and theoretically less variance in long run can be obtained.

5 1.5 Development Issues

Developing an automated trading system from scratch requires a lot of effort and resources. In the real world, there is a significant amount of APIs which are available to the public. Using these readily available APIs can reduce the cost of development significantly. However, it raises an issue of buy and build as integrating the existing APIs can be tricky and complex. Another limitation of using those APIs is that the commercial software does not connect to all the markets around the world. As a result, building our own system may be required in order to have access to multiple markets around the world. Therefore, the key question to the developer falls upon having a balance between cost and benefit as well as deciding which part of the system can be replaced by commercial APIs, which part needs customisations.

If using commercial APIs is partly the answer then this leads to another issue of data mapping. It is the process of creating blueprints and plans for converting data from format to format for communication among components in different commercial APIs. One of the proposed solutions is to create a detailed data flow descriptions and proposed structures for results. Data mapping should be developed ahead of time so that during development in code of the trading system, the integration into the existing technological environment will be well understood and well documented.

Another issue which again requires some trade-offs from the user and developer. It is between the speed of development and speed of the system. Fast systems that minimize slippage are more profitable than slower ones that may miss opportunities that exist for only milliseconds. However, faster systems may take longer to develop, may require more expensive hardware, and may require higher salary network administrators and programmers.

The development of a trading system involves not only a computer scientist but also a quantitative finance expert. As there are a lot of jargons in both fields, it is apparent that there is probability of things getting lost in translation when these two parties come together. In the real world, investment companies often employ business analysts who understand both business and the technical world. Hence they often act as interpreter between the two parties to bridge the gap of communication.

The final issue that is going to be looked at is critical to the profitability of a trading system. It is about logic leaks, or another words, a defect implementation of business logics because of miscommunication between the developers and the end users leads to incorrect representation of system functionality.

Chapter 2

Analysis

1 Statistical Arbitrage Algorithm

As the purpose of this project is primarily focused on building an automated trading system, for the purpose of simplicity, a quite simple trading strategy, statistical arbitrage, is selected to be implemented which serves mainly as a tool to test the business logic of the whole system. One important thing to note about this algorithm is that it is a very basic and popular algorithm that exists in the field of quantitative finance, hence using this strategy to make profit is unlikely to happen. Also in order to be able to build a profitable strategy, it requires much more time researching further into quantitative analytics and it is not suitable and beyond the scope of this project. The following paragraph will discuss in details the statistical arbitrage algorithm. More details of the algorithm can be found in Van Vliet, 2007.

As a trading strategy, statistical arbitrage is a heavily quantitative and computational approach to equity trading. It involves data mining and statistical methods. The trading system in development is based on a statistical arbitrage of the E-Mini S&P 500 (ES) and the E-Mini Nasdaq 100 futures contracts using a z-transform procedure.

The E-Mini S&P 500 is a simulation to the real world S&P 500 index which is a value weighted index that tracks the price of 500 largest capitalization stocks actively traded in the US (the capitalization of a company is calculated as the stock price multiply by the number of common shares issued by the company). The value weighted index is weighted against the total capitalization value of the 500 stocks. Similarly, the E-Mini Nasdaq 100 is a simulation to the stock market index of 100 of largest domestic and international non-financial companies listed on the Nasdaq stock exchange in the US. The index is also a value-weighted index against the companies' capitalization value.

A futures contract is a exchange-standardized contract to buy (long) or sell (short) a specified asset of standardized quality at a certain date (maturity date) in the future at a pre-determined price. Often, and also within this project, at the maturity date settlement of the futures contract between buyers and sellers is in cash. For example, if the system has a long (buy) position in a futures contract for 10m shares at the price of 100 for each share, and it is able to close out the position (flat) by successfully entering into the exchange a short (sell) position of a futures contract for the same amount of shares at the price of 110, the trader will make a profit of (110 -100) x 10m = 100m (price is per share).

A bid price is the price at which to buy the share, whereas an ask price is the price at which to sell the share. In the financial world, short position means sell (benefiting if index goes down in value) while long position means buy (benefiting if index goes up in value). The central point that revolves around every trading strategy is always trying to buy as low as possible and sell as high as possible.

In the following section the term "passive limit order" will be used which means an order

that has a price specified by the trader rather than the current market price. Hence it means that the order gets filled (transaction successful and becomes valid) only when the market price hits the limit price and there is also a party willing to trade at the limit price.

The details of the z-transform will be discussed in details.

Arbitrarily the price of the ratio spread is defined as followed:

$$? = \text{Pricespread} = -1 \times \text{ES Bid} + 2 \times \text{NQ Ask}$$

Based on a pre-defined set of historical prices, the normalized price is calculated as followed (in this case, the last 30 prices are taken into account), prices mentioned here and onwards are the spread price calculated as above:

$$?\text{norm} = (? - \text{MA30}(?)) / ?30\ (?)$$

- ?norm: normalized price: prices will be distributed normally, expressing the price in terms of standard deviation.
- MA_{30} is defined as the 30 spread-price-change moving average or the mean of the last 30 prices. Unlike the technical analysis system that listened for ticks (i.e., trades executing in the market), this system will only listen for changes in the bid and ask prices of the instruments. In this case, only the latest prices are obtained which in fact, price updates may be missed during the calculation of the normalized price or during the update of the user interface.
- ? is the last price.
- ?30 (?) is the standard deviation of the last 30 prices.
- A long (buy) position in the spread is defined as having 1 short (sell) position on ES futures and long (buy) 2 NQ futures contracts. A short position then is defined as long 1 ES futures contract and short 2 NQ futures contracts. If a trader has a long position overall, it means he/she is bullish on the market (speculating that the index will go up in value, hence making profit), in this case long position means bullish on value of NQ index going up as we have two long positions in NQ futures.

The strategy will be to monitor the spread price and calculate the normalized price for every price update from both NQ and ES markets. The pivotal point to note about this algorithm is the assumption that the normalized price will tend to return back to level 0 if it ever becomes greater or smaller. Hence, when the normalized price exceeds 2, we will consider this a bearish signal as we expect a reversion down to the mean, 0, and attempt to enter a short position (as mention above, this is equivalent to selling 2 NQ futures and buying 1 ES futures). Rather than simply taking the current market prices being offered, however, we will enter limit order in ES to buy 1 contract on the bid. Should we get filled (meaning there is a seller accepting our limit price and the transaction becomes valid), the system will immediately sell 2 NQ at the current market price.

Alternatively, when the normalized price is less than −2, we will consider this a bullish signal as we expect to see a reversion up the mean, 0. In this case the system will attempt to sell 1 ES on the offer. If the passive sell order is hit, we will immediately take the current market price and buy 2 NQ futures contracts.

Should we miss the trade, i.e., if the passive limit order does not get hit and the normalized

price reverts back above or below our −2 or 2 threshold, we will cancel the order and wait for another opportunity. Arbitrarily again, I have hard coded the stop price at 4 ticks. The target price will be a normalized price of 0, at which time the positions in each contract will be flattened which means we're not expecting the market prices to neither go up nor down.

Since the type of the contracts traded is futures, profit and loss amount will then be calculated based on the trades that make up the flat position of the whole portfolio. An example, if the trader makes on long futures for 10m shares at price of 100, profit or loss is calculated only if the long position is covered by a short futures (for the same amount of shares). The profit or loss will be the difference in price of the two trades multiply by the amount of shares.

2 Automated Trading System Classes

In general, objects and classes in a typical automated trading system would normally represent real world things. Since trading system development has been going on for a while, there are third party application programming interface which contains these classes. It is still important to build our own objects to facilitate our own purposes. This section will discuss typical characteristics of trading system objects currently being used.

1 Instrument Class

An instrument object represents a tradable security or financial instrument. The design of an instrument class will be concerned with the frequent changes to the class' data members such as bid and ask prices change, trades go off, and the quantities to buy and sell on the different price levels change constantly. But not all updates on the instrument are required by the algorithm of the system. Therefore it is important to decide what frequency of updates to the instrument class is suitable for a particular trading algorithm. For example, with the statistical arbitrage algorithm, we may only be interested in the latest bid and ask prices. Having an update filter built into the system would be beneficial.

2 Order Class

An order class represents a working order at the exchange. In general, the data members of an order often contain the following features: an unique identifier, whether the order is a buy or a sell one, the quantity, the limit or stop loss price, and the size of the instrument's tick.

When an order is sent to the exchange, what usually happens to the order is that it can be filled or partially filled by the exchange. It can also be cancelled by the trader when market conditions change or cancelled after a particular time interval if the limit or target price has not been hit. Another possibility is that the order could even be rejected by the exchange. Careful research needs to be carried out to maintain the best order management logic. Every possible outcome must be considered before the implementation of the trading system can start.

In fact, having a limit order at a position at or near the front of a long queue in a high frequency trading system is valuable. Optimizing order management algorithms for partial fills and time and volumes can make the difference between success and failure. (Van Vliet, 2007).

3 Order Book Class

An order book class is often a collection of orders which keeps track of all valid orders. As

orders often have unique identifier, designing the order book as a hashtable is often a good solution. In the case where orders are sequential, SortedList is the answer.

4 Bracket Class

A bracket class consists of a target price and a stop price for an open position. Each bracket class needs to be signalled when there are price changes in the market, or when the target or limit price has been hit so that required action to the order can be made. Stop prices in a bracket class can be made to be trailing stops so as the price of the instrument moves in favour of our position, the stop price is raised.

5 Tick Class

A tick is the minimum upward or downward movement in the price of a financial instrument. Often data members of a typical tick class contain price, quantity (could mean number of shares or index points) and time.

6 Tick Collection Class

A tick collection is often a collection of all the latest ticks received from the market.

7 Bar Class

A bar is often used in charts. It is a graphical representation of a stock's movement that usually contains the open, high, low and closing prices for a set period of time.

8 SystemManager Class

This is often referred to as the main thread in a trading system which controls the flow of the application and also the business logic. It also is responsible for maintaining connectivity with the exchange, handling updates from the market, order and position management and trade selection of the trading system.

When the trading algorithm is long and complex, it is often a good strategy to segregate trade selection, order management and risk management logic into separate processes. Also handling database connectivity should be separated from the main thread as well.

9 Graphical User Interface

The user interface is often the place to display important data and current states of the trading system including real time market prices, position updates, network connection alerts and so on. It should also consist of controls which let the trader interact with the application.

Updating the user interface often leads to adverse effect on the overall performance of the system, so a separate thread with a low priority to handle the interface should be used.

3 traderAPI Package

More details on this package could also be found in Van Vliet, 2007. This section will introduce and touch on the basic characteristics of the package and the role it plays within the system.

traderAPI is a .NET dll that emulates the XTAPI offered by Trading Technologies, Inc. XTAPI is a futures trading platform which facilitates market connectivity worldwide. Unlike XTAPI, traderAPI, although mimics the behaviour of XTAPI, it has an inner price generator to simulate the market price changes. Therefore it is important to note that outcome from this trading system does not reflect what will happen in the real world market.

TraderAPI has a random, internal price generation mechanism for E-Mini S&P 500, symbol ES, and E-Mini Nasdaq 100, symbol NQ, futures contracts traded on the Chicago Mercantile Exchange. Details of all the classed provided in the traderAPI package can also be found in Van Vliet, 2007.

4 Technology Choices

Visual C++ within the .NET framework is the technology of choice. Visual C++ has strengths that will best facilitate requirements of a typical automated trading system development. Moreover the .NET framework contains many built-in classes that will aid the development of the system.

- In financial market and trading system development, interoperability is key. Relative to other languages, Visual C++ contains much greater support for interoperability between managed and unmanaged code.
- In Visual C++.NET, we have precise control over managed and unmanaged heap and stack memory allocation and managed and unmanaged code and all can exist seamlessly in the same project. This enables the developers to quickly and easily integrate third party packages into the application.
- Also for a long time, C++ alone has been popular for best in data-intensive tasks which suits the design purposes of this project.
- Using visual studio 2005 is beneficial in building the graphical user interface with the built-in drag and drop features.

Chapter 3

Design

This trading system is based on a statistical arbitrage of the E-Mini S&P 500 (ES) and the E-Mini Nasdaq 100 futures contracts. Since it is automated, it is vital that the quantitative part of the system (business logic) is carried out without any disruption. Moreover, as the outcome of the strategy evaluation depends on the historical prices, the application needs to be free to receive all the price updates from the market. Therefore, it is decided that a multithreaded system is the best option that can facilitate the dynamics required by the application. This application will employ the Instrument class which incorporates the Producer/Consumer pattern. Their characteristics and how they fit in the whole system will be discussed in the following sections.

We will use one thread with the highest priority to evaluate all the calculations (calculating the normalized price, the spread price and evaluation of the statistical arbitrage trading strategy) to produce the optimal position for the trader based on the historical prices. There are two threads that will be responsible for connections and receiving price updates from the traderAPI simulation. The main thread which runs when the application is initiated creates the user interface and also is responsible for all the updates made to the interface.

3.1 Instrument Class

Within the context of building a trading system, an instrument object represents a financial instrument which can be either a tradable security or a derivative. In fact many of these objects maybe contained within a third-party package such as the traderAPI package in use, it is likely the optimal solution to create our own object for the purpose of future modifications and customizations.

When making a trade or an order to the exchange, the instrument will go back and forth between the exchange and the trader's platform, it is decided that all the complexities of making connections to traderAPI are encapsulated within this instrument class. This way we will have more control over the program flow and makes life easy when it comes to implementation of the system and should a new version of the API be released, there is no need to tear apart the whole application.

Typical details of a financial instrument are encapsulated within this class such as the tick size, the bid and ask prices as well as "get" methods to access those data. Moreover, there are events handlers to handle all the updates to the instrument sent from the exchange. In a trading environment, bid and ask prices are constantly changing, trades go off, and the quantities to buy or sell on the different price levels change constantly. As we are not concerned with getting every tick, only the latest 30 bid/ask prices, the capability of traderAPI which allows us to filter updates from the exchange is employed. The multicast update OnPriceUpdate and OnFill delegates allow

for this.

Following is the table which summarizes all the properties and methods encapsulated in the instrument class. Details of how connections to traderAPI are made will be discussed in the implementation chapter.

Methods	**Descriptions**
OnNotifyUpdate	Fires when there are updates to the instrument object
OnOrderFillData	Fires when the order from the trader is filled
EnterMarketOrder	Create and send an order to the exchange. Returns a boolean value.
CancelOrder	Cancel the order. Returns a boolean value.
EnterLimitOrder	Sends a limit order to the exchange. Returns an order object to our instrument class
get_Bid	Get the bid price. Returns int.
get_Ask	Get the ask price. Returns int.
get_TickSize	Get the tick size. Returns int.
get_NetPos	Get the net position. Returns int.

Events	**Descriptions**
FillEventHandler	Handles the fill notification from the exchange
PriceUpdateEventHandler	Handles all the price updates

Properties	**Descriptions**
m_TickSize	Size of one tick
m_Bid	Bid price
m_Ask	Ask price

3.2 Other System Classes

Besides the instrument class which plays the central and vital role in an automated trading system, there are other classes that will make up the entire system. The following sections will discuss how they fit into the system in more details.

3.2.1 Order Class

An order in a trading system represents a working order at the exchange. For the purpose of simplicity, the application's order will contain only the basic characteristics of a typical real world order. In fact, a trading system often has very high frequency of orders placed by the trader or the system itself, even in a low volatility market. Hence, it is favorable in this context to use limit order rather than market order. Also it is valuable to have a limit order at a position at or near the front of a possible long queue of standing orders. As it is a limit order, data members will include the unique identifier which is of type "String" and the limit price which is of type "int".

3.2.2 Order Book Class

The order book of the application is a collection of orders. Two order books are created; one to store all the buy orders, the other one will store all the sell orders. The two collections are of type "SortedList" which will be sorted in the order of the time they are created.

3.2.3 Tick Class

The tick class of this application will consist of three data members. They will store the time, price and quantity (or volume).

3.2.4 Tick Collection Class

The tick collection class of this application will be a list which stores the last 30 bid/ask price changes from the exchange. The list is of type "ArrayList" which is able to facilitate the expansion of the list itself in the future in case a different trading algorithm is put in place that might require more past ticks to be stored.

3.3 Producer/Consumer Design Pattern

Basically, this multithreaded trading system contains three threads. One thread plays the role of the producer which is the exchange, one thread plays the role of the consumer which is the trader and the other thread is to do with the user interface. In this section, we will discuss this producer and consumer pattern. The exchange, or the producer, generates data (prices), and the consumer, or the trader, consumes that data.

The producer and consumer share data through a common buffer, a memory location. The two actors, though, produces and consumes data at different rates. This brings up an issue of whether the consumer will get "real time" data from the producer. In the real world, price changes happen constantly which likely to create latency in the system. Moreover, during the computation process, complex algorithms could take up a long time which would force the trader's system to lag behind. Therefore, the striking question for developers of automated trading system is whether they are willing to forgo some data to ensure the current data is as up to date as possible. In this particular context, our application employs statistical arbitrage algorithm which only requires the latest 30 bid/ask prices, it is not necessarily critical that we catch all market data; we are only concerned about the most recent price changes from the market.

With the "get the latest ticks" design, a shared semaphore is used. The consumer thread

will wait for the producer thread to generate data. When data is generated, the consumer thread receives the signal from the producer; it calls the get data method for retrieval. In case the consumer is busy consuming the data when the signal arrives, the producer simply overwrites the previous data. This way, the consumer is always guaranteed to receive the most up to date information. At times when the consumer is faster than the producer, causing an update when none is necessary. This problem is resolved by having the consumer thread first check the data ensuring that the data is the latest before proceeding.

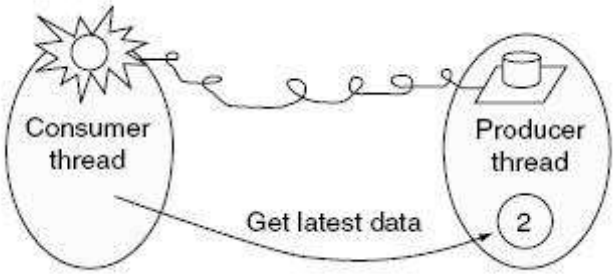

Figure 3.1: Producer/Consumer design pattern **(**Van Vliet, 2007**)**

To summarize this producer/consumer pattern is important to general automated trading system development since such systems contain threads that are receiving real time market data for consumption by other threads that encapsulate trade selection, order management, and risk management algorithms.

3.4 The Multithreaded System

Unlike other single-threaded system which performs all the processes in one single thread, this system will run the user interface on its own thread, perform all the algorithm calculations on another thread, and real time data generators (the two instrument classes) each on its own thread. In this system, which method running on which thread is not always clear. When the trading application is initiated by the user, a new thread, the Main Thread, starts with a new instance of the user interface. The SystemManager class is created when connections to the data generator are made. The other two threads, RTDThread_A and RTDThread_B, run applications with new instances of the instrument class: one for ES exchange (thread A and instrument A), the other one is for NQ exchange (thread B and instrument B).

The thread on which the SystemManager class runs will control all the business logic which carries out all the quantitative analysis. Then it sends all post-processed data to the user interface. While the instruments are running on their own RTD threads, the responsibilities of entering limit and market orders lie on the Strategy Thread. When the instrument A or instrument B receives a fill update from the traderAPI, they will raise the corresponding events which are handled by handlers of the SystemManager class. These handling methods will execute on the corresponding RTDThread_A or RTDThread_B. Updates that need to be made to the user interface is carried out by a thread with the lowest priority giving the thread that performs all the quantitative analysis on data the highest priority. This will ensure the least possible disruption to the calculation process. With reliability and stability, the system would hopefully not miss any opportunity in the market which could only exist for a few seconds or even less.

The following figure shows how all the threads and classes fit into the whole system. The CalProcessor refers to the engine which carries out all the necessary quantitative analysis.

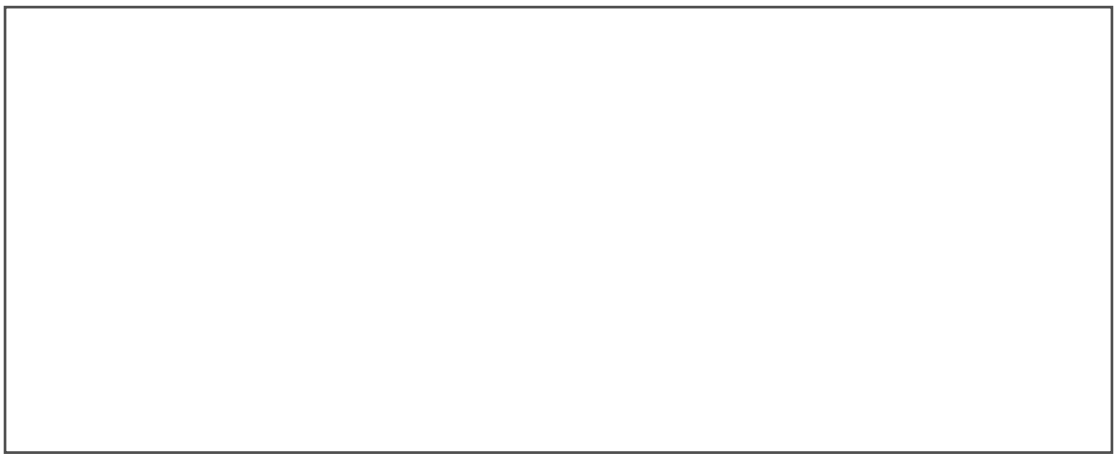

Figure 3.2: Overall system design

How all the above mentioned threads work with each other is illustrated in the following figure. The Form 1 in the figure refers to the user interface screen which pops up when the application is first initiated. The Main Thread is the one that is first created and deals with the UI, so has the lowest priority.

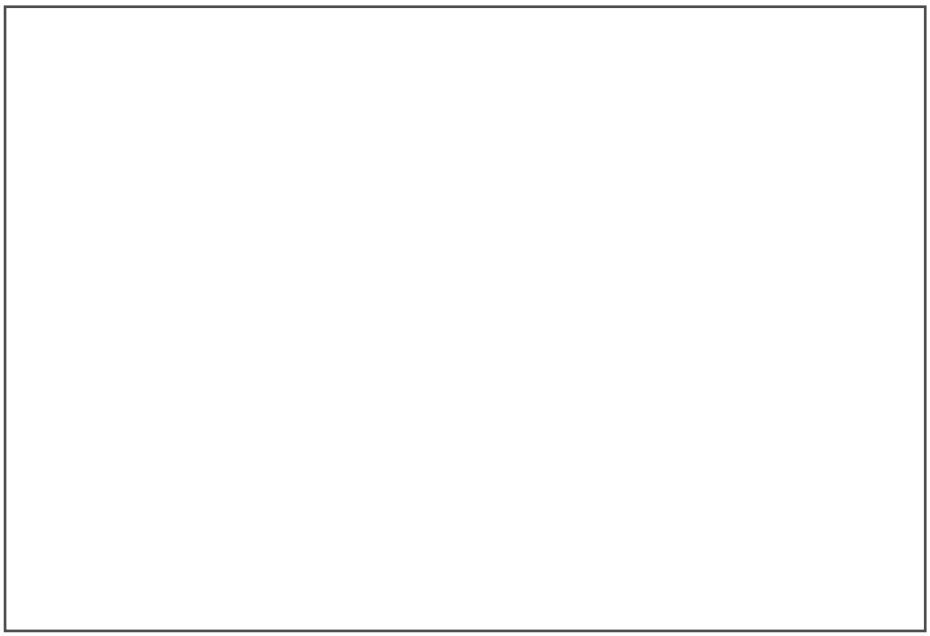

Figure 3.3: Application threads

Chapter 4

Implementation

4.1 Graphical User Interface

As it is also for the entire system design, for the purpose of simplicity, inputs and changes from the user or trader to the user interface are not allowed. The user interface will show the real time price updates, including the spread price, the positions and the stop prices. The dataGridView object will show all the transactions that have been placed successfully with the exchanges. There's also a profit and loss view which records the up to date profit or loss. The profit and loss is calculated based on valid trades, ones which make the overall position FLAT, neither short or long overall. The following figure will illustrate the design of the user interface.

A text area is also dedicated as a log to record all the traffic between the system and the exchanges including events such as entering trades to the exchange, fill events from the exchanges, order cancellations. The time of the log is the local time of the machine which this trading system is run on.

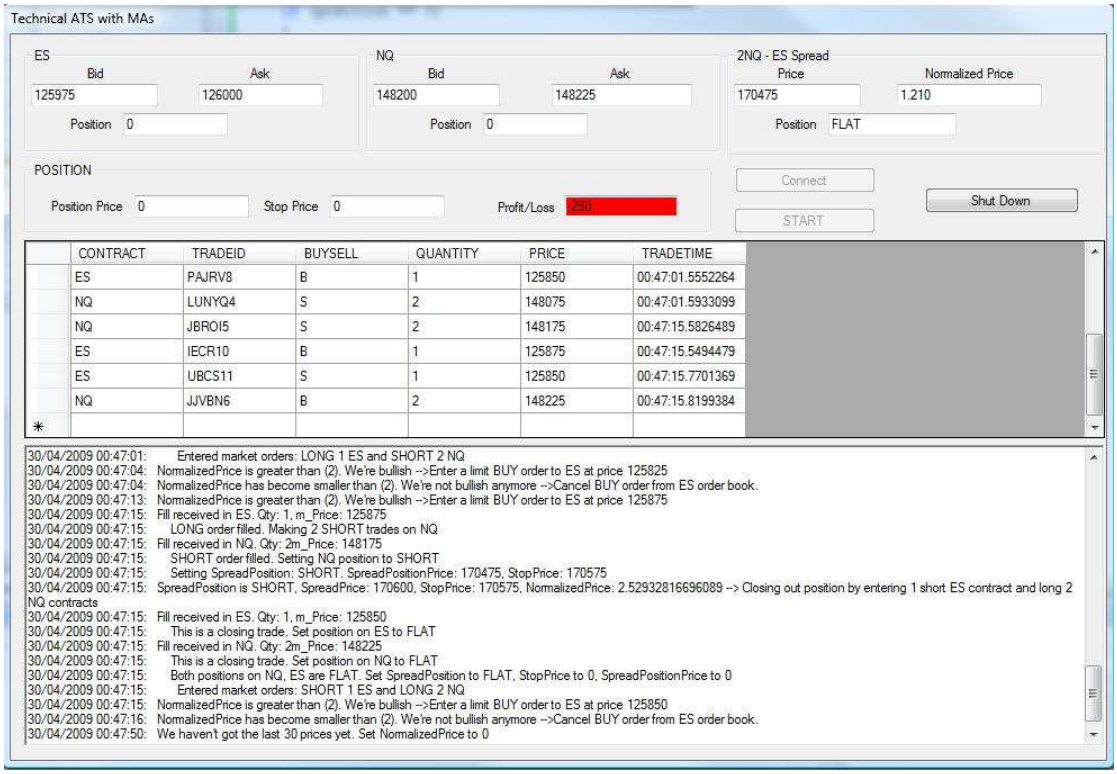

Figure 4.1: Graphical User Interface

The above screen together with two exchanges views are the screens that make up the whole user interface. This interface will mainly show the results of the quantitative analysis performed behind the scene by a different thread. Apart from all the text boxes that show the bid, ask prices and positions which are self-explanatory, the following part of this section is focused on describing the implementation of the three buttons: CONNECT, GO and SHUT DOWN. The system is designed in the way that it is not interactive with the user apart from the user's ability to start and stop the system from running. As the system is automated, it will make the decisions to enter trades into the exchanges automatically based on the statistical arbitrage strategy. The profit and loss is calculated in real time, the number gets updated every time the overall position of the portfolio becomes flat (total number of short contracts is equal to that of long contracts, in both exchanges). If it is a loss, the textbox will turn red, otherwise it is green indicating that we are making a profit overall.

The trading system is started with the user pressing the CONNECT button on the screen to initiate the two instrument threads to simulate the two exchanges ES and NQ. This action also initializes the system data which is used to store details of all the transactions.

```
|private: System::Void button1_Click(System::Object^  sender,              |
|System::EventArgs^  e)                                                    |
|{                                                                         |
|// Create a dataset to keep fill information                              |
|try                                                                       |
|{                                                                         |
|m_FillData = gcnew DataSet;                                               |
|Log("Setting up DataSet.");                                               |
|SetUpDataSet();                                                           |
```

```
|// Create new SystemManager object and subscribe to delegates.           |
|Log("Creating new SystemManager and subscribing to delegates:");         |
|m_Manager = gcnew SystemManager(this);                                   |
|Log("SystemManager created");                                            |
|m_Manager->OnInstrumentPriceUpdate += gcnew UpdateEventHandler(          |
|this, &Form1::OnPriceUpdateEventHandler);                                |
|m_Manager->OnSystemFillUpdate += gcnew SystemFillEventHandler(           |
|this, &Form1::OnFillEventHandler);                                       |
|m_Manager->Log += gcnew LogHandler(                                      |
|this, &Form1::Log);                                                      |
|// Set buttons for secure start up/shut down.                            |
|button1->Enabled = false;                                                |
|button2->Enabled = true;                                                 |
|this->ControlBox = false;                                                |
|}                                                                        |
|catch (Exception ^e)                                                     |
|{                                                                        |
|Log("Exception occured: " + e->ToString());                              |
|}                                                                        |
|}                                                                        |
```

The SystemManager thread which controls updates to the GUI and the running of the strategy thread is initiated by the START button. The two handlers will be created and be responsible for handling updates from the two instrument classes and fill notifications from the exchanes. After START is clicked, the two buttons START and CONNECT will be disabled so that the simulation can progress without being prompted to initialize again.

```
|private: System::Void button2_Click(System::Object^  sender,            |
|System::EventArgs^  e)                                                  |
|{                                                                       |
|// profit and loss is 0 initially                                       |
|profitLoss = 0;                                                         |
|position = 0;                                                           |
|                                                                        |
|// Start the trading system.                                            |
|Log("Start button is clicked");                                         |
|m_Manager->Go();                                                        |
|button2->Enabled = false;                                               |
|this->button3->Enabled = true;                                          |
|}                                                                       |
```

The user presses SHUT DOWN to shut down the connections made to the simulator. Then the user will be able to close the application. In order to clear the memory, the SystemManager will be deleted from the memory together with its two handlers. After that, the garbage collector is called to free up the memory of the computer which is occupied by the system.

```
|private: System::Void button3_Click(System::Object^  sender,            |
|System::EventArgs^  e)                                                  |
|{                                                                       |
|// Shut down system gracefully.                                         |
|if (m_Manager != nullptr)                                               |
|{                                                                       |
|delete m_Manager;                                                       |
|m_Manager->OnInstrumentPriceUpdate -= gcnew UpdateEventHandler(         |
```

```
|this, &Form1::OnPriceUpdateEventHandler);                                    |
|m_Manager->OnSystemFillUpdate -= gcnew SystemFillEventHandler(               |
|this, &Form1::OnFillEventHandler);                                           |
|m_Manager = nullptr;                                                         |
|GC::Collect();                                                               |
|}                                                                            |
|this->ControlBox = true;                                                     |
|}                                                                            |
```

4.2 Instrument Class

Within this instrument class, apart from all the data members which are characteristics of a financial instrument such as tick size, bid and ask prices, the connection to the traderAPI is also encapsulated. This part is done in the class constructor. First, an instrument object and an instrument notify class is created. The instrument notify class allows us to be able to receive price updates and to turn on the price update filter. [pic]

Next, set the exchange, product and contract type. The financial product used for the purpose of illustration is futures instrument.

```
|m_Instr->Exchange = "CME-SIM"; // the name of the exchange,          |
|specified by the simulation                                         |
|m_Instr->Product = m_P;                                             |
|m_Instr->Contract = m_C;                                            |
|m_Instr->ProdType = "FUTURE";                                       |
```

An order set class is created with limits if violated will shut down the application automatically. This class, besides, allows the simulation to delete orders placed by the application.

```
|m_OrderSet = gcnew OrderSetClass;                                   |
|m_OrderSet->set_Set("MAXORDERS", 1000); // maximum number of orders|
|in the order set                                                    |
|m_OrderSet->set_Set("MAXORDERQTY", 1000); // maximum quantity per  |
|order at 1000 index points                                          |
|m_OrderSet->set_Set("MAXWORKING", 1000);                            |
|m_OrderSet->set_Set("MAXPOSITION", 1000);                           |
|m_OrderSet->EnableOrderAutoDelete = true;                           |
|m_OrderSet->EnableOrderFillData = true;                             |
|m_OrderSet->EnableOrderSend = true;                                 |
```

There are two handlers within this class. One of them is invoked when there are updates to the instrument class itself. The other handler is fired when it is notified that the order has been filled.

The EnterMarketOrder function will try to create an order profile which stores all the information about the market order which is to be sent to the exchange. It returns a Boolean value to indicate that whether the market order has been successfully created and sent.

The CancelOrder function when called upon will cancel the order sent to the exchange by the system previously. It also returns a Boolean value to indicate whether the order has been successfully cancelled.

[pic]

The EnterLimitOrder allows the application to enter a limit order which could be a stop loss order to the exchange. It returns an order object to the instrument class.

[pic]

4.3 SystemManager Class

The SystemManager class will control all the business logic, calling the static method that carries out all the calculations and is responsible for updating the user interface.

Within the constructor, before performing all the business logic, two threads, RTDThread_A and RTDThread_B, are created together with two order books (sell book and a buy book). Also the tick collection is created here too. The starting positions for a trading application are always FLAT.

After the class is created, it also starts the two threads which run two instruments.

Upon receiving the signal that there are good bid/ask data available, the SystemManager will grab the new data and calculate the normalized price. It then checks to see if a stop or target has been hit. If we are working an order and the normalized price has already crossed back over the threshold, the working order will be cancelled. The strategy thread will then next check to see if there is a new indication to enter an order. Finally the user interface is updated. All these activities and calculations are performed on a separate thread, the run strategy thread, which basically plays the role of implementing the statistical arbitrage system when there are price updates from the exchanges.

[pic]

4.4 Other Classes

4.4.1 Order Class

A simple order class that contains of all the information needed about a working limit order. It consists of a unique identifier number and the price.

4.4.2 Tick Class

A tick contains 3 data members which make up an actual tick. They are time, quantity and price.

4.4.3 TickData Class

This class contains the semaphore which is used to prevent cross thread access to the price variables.

4.4.4 NormCalc Class

This class contains the method which performs the calculation of the normalized price based on a list of the last 30 historical spread prices.

[pic]

Chapter 5

Testing

5.1 System testing

The functionalities of the system are tested by carrying out probationary trades. The market data is generated using a price generated mechanism built-in the traderAPI package. An important point to note about the testing that for the purpose of simplicity, the system has been designed to not be interactive with the user which reduces significantly unexpected data or events; hence it makes it easier to identify the source and handle exceptions during run time.

Within the scope of this project, functional testing has been determined to include ensuring that trades that are submitted to the simulated exchanges strictly adhere to the statistical arbitrage algorithms. This testing phase was carried out by checking the traffic log. Thus the implementation of this testing phase is performed manually.

The fundamental events that should happen include the system making a short position (long 1 ES (limit order) and short 2 NQs (entering these trades by taking the market price only if the long position in ES gets filled)) when the normalized price exceeds 2 and a long position (short 1 ES and long 2 NQs) when it becomes smaller then (-2). Another important event from the system is the cancellation of buy or sell orders if they have not got filled and the normalized price turns back to the range between -2 and 2. The process was performed three times manually and this stage's status was considered to be passed.

Further to the functional testing, the calculation of the normalized price also needs to be tested thoroughly as having the correct normalized price is pivotal to the success of implementation of the statistical arbitrage trading strategy. This was also performed manually by checking the return value updated on the GUI. The status was also passed.

5.2 Suggested Future testing

Due to the fact that the market data is obtained from a simulator, possible future testing of system functionalities could include:

- Using real market data by making connections to the real world markets. It should be noted that the market is designed for E-Mini S&P 500 (ES) and the E-Mini Nasdaq 100 futures contracts.
- Use a more complex trading strategy. This way the volume and the frequency of trades going in and out of the system increased significantly so that we can test the reliability of the system.
- Consistency of the system can be tested by running the system through different market conditions such as market is booming, stagnating or in recession.

Chapter 6

Conclusion

6.1 Overall Conclusion

Overall, project is considered to be complete. The architecture for an automated trading system is achieved and successfully implemented. The learning part of Visual C++ is also complete. A good understanding of Visual C++ and the .NET framework has been achieved. More importantly this project is a good practice of going through the trading system development methodology.

However, the testing phase has not been performed as thorough as one might have wished. A market simulator is employed which did not reflect the real market hence it is difficult to assess the true performance of the system.

In addition, this complete system would become an useful starting point for further research into the field of automated trading system for both computer scientists and quantitative finance.

6.2 Future Improvement

- Expand the system so that it can facilitate a wider range of financial instruments. This system uses futures contract as the standard instrument. The system can be improved to be able to connect to markets worldwide and also can be used to trade other derivatives or equity or bond.
- Profitability is critical to an automated trading system; hence further research can be put in to the quantitative finance field so that profitable and suitable strategies can be developed for appropriate instruments and markets.

Chapter 7

References

Mark R. Conway, Aaron N. Behle. Professional Stock Trading: *System Design and Automation*. Acme Trader, 1st Edition.

Mark Jurik. Computerized Trading: *Maximizing Day Trading and Overnight Profits*. New York Institute of Finance. Prentice Hall, 1999.

Murray A. Ruggiero, Jr.. Cybernetic Trading Strategies: *Developing a Profitable Trading System with State-of-the-Art Technologies*. John Wiley and Sons, 1997.

Chapter 8

Bibliography

Benjamin Van Vliet. Building Automated Trading Systems: *With An Introduction to Visual C++ .NET 2005*. Elsevier, 2007.

Perry J. Kaufman. Trading System and Methods. Wiley, 1998.

Perry J. Kaufman. Smarter Trading: *Improving Performance in Changing Markets*. McGraw Hill, 1995.

```
// Open the connection to TraderAPI.
        m_Instr = gcnew InstrObjClass;
        m_Notify = dynamic_cast <InstrNotifyClass^> (m_Instr->CreateNotifyObj);
        m_Notify->EnablePriceUpdates = true;
        m_Notify->UpdateFilter = m_Filter;
        m_Notify->OnNotifyUpdate += gcnew InstrNotifyClass::OnNotifyUpdateEventHandler
                        (this, &Instrument::OnNotifyUpdate);



void Instrument::OnNotifyUpdate(InstrNotifyClass ^pNotify,
                                InstrObjClass ^pInstr)
{
        // When there is an update this function is invoked, in this case only the recent tick  is
used.
        Object ^m_Data = pInstr->get_Get("BID, ASK");

        m_Bid = Convert::ToInt32(safe_cast< array< String ^ >
                                >(m_Data)[0]);
        m_Ask = Convert::ToInt32(safe_cast< array< String ^ >
                                >(m_Data)[1]);

        TickData::ConsumerSemaphore->Release(1); // Release the lock on the semaphore so
that other threads can use them
}

void Instrument::OnOrderFillData(FillObj ^m_Fill)
{
        array< String ^ > ^ FillData = safe_cast< array< String ^ >
      ^ >(m_Fill-get_Get("PRODUCT,KEY,BUYSELL,QTY,PRICE,TIME"));

        OnFill(FillData[0], FillData[1], FillData[2],
                Convert::ToInt32(FillData[3]),
                                Convert::ToInt32(FillData[4]),
                               FillData[5]);
}




bool Instrument::EnterMarketOrder(String ^m_BS, int m_Qty, String ^m_FFT)
{
        try
        {
                OrderProfileClass ^m_Profile = gcnew OrderProfileClass;
                m_Profile->Instrument = m_Instr;
                m_Profile->set_Set("ACCT", "12345"); // account number, which is insignificant in
this case
                m_Profile->set_Set("BUYSELL", m_BS);
                m_Profile->set_Set("ORDERTYPE", "M");
                m_Profile->set_Set("ORDERQTY", m_Qty.ToString());
```

```
			m_Profile->set_Set("FFT3", m_FFT);
			__int64 m_Result = m_OrderSet->SendOrder(m_Profile);
			return true;
		}
		catch(Exception ^e)
		{
			return false;
		}
}


bool Instrument::CancelOrder(String ^m_Key)
{
		try
		{
			m_OrderSet->Cancel(m_Key);
			return true;
		}
		catch(Exception ^e)
		{
			return false;
		}
}


Order ^Instrument::EnterLimitOrder(String ^m_BS, int m_Px, int m_Qty, String ^m_FFT)
{
		// This method will run on the Strategy thread.
		try
		{
			OrderProfileClass ^m_Profile = gcnew OrderProfileClass;
			m_Profile->Instrument = m_Instr;
			m_Profile->set_Set("ACCT", "12345");
			m_Profile->set_Set("BUYSELL", m_BS);
			m_Profile->set_Set("ORDERTYPE", "L");
			m_Profile->set_Set("LIMIT", m_Px.ToString() );
			m_Profile->set_Set("ORDERQTY", m_Qty.ToString() );
			m_Profile->set_Set("FFT3", m_FFT);
			__int64 m_Result = m_OrderSet->SendOrder(m_Profile);
			return gcnew Order(m_Profile->get_GetLast("SITEORDERKEY"), m_Px);
		}
		catch (Exception ^e)
		{
			return nullptr;
		}
}


SystemManager::SystemManager(Form ^m_F)
```

```
{
	m_Form = m_F;
	// Start RTD Thread for instrument A (ES).
	m_RTDThread_A = gcnew Thread(gcnew ThreadStart(this,
&SystemManager::RunRTD_A));
	m_RTDThread_A->Priority = ThreadPriority::Highest;
	m_RTDThread_A->Name = "RTD Thread A";
	m_RTDThread_A->Start();


	// Start RTD Thread for instrument B (NQ).
	m_RTDThread_B = gcnew Thread(gcnew ThreadStart(this,
&SystemManager::RunRTD_B));
	m_RTDThread_A->Priority = ThreadPriority::Highest;
	m_RTDThread_B->Name = "RTD Thread B";
	m_RTDThread_B->Start();

	// SortedLists will keep track of working orders in the market.
	m_BuyOrderBook = gcnew SortedList;
	m_SellOrderBook = gcnew SortedList;
	// An ArrayList will keep track of the last 30 bid/ask changes.
	m_PriceList = gcnew ArrayList;
	// Start with flat positions, obviously.
	m_SpreadPos = Position::FLAT;
	m_Pos_A = Position::FLAT;
	m_Pos_B = Position::FLAT;
}



void SystemManager::RunRTD_A()
{
	try
	{
		// Create new Instrument object and subscribe to the fill event.
		m_Instrument_A = gcnew Instrument("ES", "Sep08",
                                                                "BID,ASK");
		m_Instrument_A->OnFill += gcnew FillEventHandler(this,
                                &SystemManager::OnFill_AEventHandler);

		// Start a new message loop for RTD for instrument A (ES).
		Application::Run(m_Instrument_A);
	}
	catch (Exception ^e)
	{
		Debug::WriteLine(e->Message);
	}
} // end of the RunRTD_A method.

void SystemManager::RunRTD_B()
{
```

```
	try
	{
		// Create new Instrument object and subscribe to fill event.
		m_Instrument_B = gcnew Instrument("NQ", "Sep08", "BID,ASK");
		m_Instrument_B->OnFill += gcnew FillEventHandler(this,

&SystemManager::OnFill_BEventHandler);

		// Start a new message loop for RTD for instrument B (NQ).
		Application::Run(m_Instrument_B);
	}
	catch (Exception ^e)
	{
		Debug::WriteLine(e->Message);
	}
}



// Have we hit a stop or have we hit our target? If so, close positions.
		if (m_SpreadPos == Position::LONG && (m_SpreadPrice <= m_StopPrice ||
m_NormPrice > 0))
		{
			m_Form->BeginInvoke(Log,"SpreadPosition is LONG, SpreadPrice: " +
m_SpreadPrice.ToString() + ", StopPrice: " + m_StopPrice.ToString() +
				", NormalizedPrice: " + m_NormPrice.ToString() +
				" --> Closing out position by entering 1 long ES contract and short
2 NQ contracts");

			bool m_Bool = m_Instrument_A->EnterMarketOrder("B", 1, "CLOSE");
			m_Bool = m_Instrument_B->EnterMarketOrder("S", 2, "CLOSE");

			m_Form->BeginInvoke(Log, "	Entered market orders: LONG 1 ES and
SHORT 2 NQ");
		}
		if (m_SpreadPos == Position::SHORT && (m_SpreadPrice >= m_StopPrice ||
m_NormPrice < 0))
		{
			m_Form->BeginInvoke(Log,"SpreadPosition is SHORT, SpreadPrice: " +
m_SpreadPrice.ToString() + ", StopPrice: " + m_StopPrice.ToString() +
				", NormalizedPrice: " + m_NormPrice.ToString() +
				" --> Closing out position by entering 1 short ES contract and long
2 NQ contracts");

			bool m_Bool = m_Instrument_A->EnterMarketOrder("S", 1, "CLOSE");
			m_Bool = m_Instrument_B->EnterMarketOrder("B", 2, "CLOSE");

			m_Form->BeginInvoke(Log, "	Entered market orders: SHORT 1 ES
and LONG 2 NQ");
		}
```

```
                // If the reason for buying or selling the spread no longer exists,
                // cancel the working order. i.e. if we missed the trade, cancel it.
                if (m_NormPrice < 2 && m_BuyOrderBook->Count > 0)
                {
                        m_Form->BeginInvoke(Log,"NormalizedPrice has become smaller than
(2). We’re not bullish anymore -->" +
                                "Cancel BUY order from ES order book.");

                        // Cancel buy order and remove from order book.
                        m_Instrument_A->CancelOrder(safe_cast< Order ^ >(m_BuyOrderBook-
>GetByIndex(0))->get_Key());
                        m_BuyOrderBook->RemoveAt(0);
                }
                if (m_NormPrice > -2 && m_SellOrderBook->Count > 0)
                {
                        m_Form->BeginInvoke(Log,"NormalizedPrice has become smaller than (-
2). We’re not bearish anymore -->" +
                                "Cancel SELL order from ES order book.");

                        // Cancel sell order and remove from order book.
                        m_Instrument_A->CancelOrder(safe_cast< Order ^ >(m_SellOrderBook-
>GetByIndex(0))->get_Key());
                        m_SellOrderBook->RemoveAt(0);
                }

                // Make a decision as to whether or not to enter a trade.
                // Make a long trade in A if normalized price > 2 and we are
                // flat and not already working an order.
                // Enter order method calls run on the Strategy thread.
                if (m_NormPrice > 2 && m_Pos_A == Position::FLAT && m_BuyOrderBook-
>Count == 0)
                {
                        m_Form->BeginInvoke(Log,"NormalizedPrice is greater than (2). We’re
bullish -->" +
                                "Enter a limit BUY order to ES at price " + m_Bid_A.ToString());

                        // Try to buy 1 on the bid.
                        Order ^m_Order = m_Instrument_A->EnterLimitOrder("B", m_Bid_A, 1,
"OPEN");
                        m_BuyOrderBook->Add(m_Order->get_Key(), m_Order);
                }

                // Make a short in A if normalized price < "2 and we are flat and not alr// Make a
short in A if normalized price < −2 and we are flat and not already
                // working an order.
                if (m_NormPrice < -2 && m_Pos_A == Position::FLAT && m_SellOrderBook-
>Count == 0)
                {
                        m_Form->BeginInvoke(Log,"NormalizedPrice is smaller than (-2). We’re
bearish -->" +
                                "Enter a limit SELL order to ES at price " + m_Ask_A.ToString());
```

```
                        // Try to sell 1 on the ask.
                        Order ^m_Order = m_Instrument_A->EnterLimitOrder("S", m_Ask_A, 1,
"OPEN");
                        m_SellOrderBook->Add(m_Order->get_Key(), m_Order);
                }

                // Update the form on the main thread.
                m_Form->BeginInvoke(OnInstrumentPriceUpdate);


TickData::ConsumerSemaphore->WaitOne();
                // Get latest Bid/Ask data.
                m_Bid_A = m_Instrument_A->get_Bid();
                m_Ask_A = m_Instrument_A->get_Ask();
                m_Bid_B = m_Instrument_B->get_Bid();
                m_Ask_B = m_Instrument_B->get_Ask();

                // Calculate the spread Bid/Ask for 1×2 and add it to the list.
                m_SpreadPrice = 2 * m_Ask_B - m_Bid_A;
                m_PriceList->Add(m_SpreadPrice);

                // Calculate the normalized price.
                m_NormPrice = NormCalc::CalcNormalizedPrice(m_PriceList);

                if (m_NormPrice == 0)
                {
                        m_Form->BeginInvoke(Log, "We haven’t got the last 30 prices yet. Set
NormalizedPrice to 0");
                }




ref class Order
{
        private:
                String ^m_Key;
                int m_Price;
        public:
                Order(String ^key, int price) : m_Key(key), m_Price(price) {}
                String ^get_Key() {return m_Key;}
                int get_Price(){return m_Price;}
};



value struct Tick
{
        double Price;
        int Volume;
        DateTime Time;
```

```
};


ref class NormCalc
{
	public:
		static double CalcNormalizedPrice(ArrayList ^m_List)
		{
			// This method will run on the Strategy thread.
			double m_Mean = 0;
			double m_StDev = 0;
			if (m_List->Count == 30)
			{
				// Calculate the 30 tick MA.
				for(int x = 0; x < 30; x++)
				{
					m_Mean += Convert::ToDouble(m_List[x]);
				}
				m_Mean /= 30;
				// Calculate the 30 tick St Dev.
				for(int x = 0; x < 30; x++)
				{
					double m_priceMeanDifference =
Convert::ToDouble(m_List[x]) - m_Mean;
					m_StDev += Math::Pow(m_priceMeanDifference, 2);
				}
				m_StDev = Math::Sqrt(m_StDev/30);

				// remove the oldest price in the list (which is at the first of the list)
				m_List->RemoveAt(0);

				// Return (LastPx – 30 tick MA)/30 tick St Dev
				double m_difference = Convert::ToDouble(m_List[28]) - m_Mean;
				return (m_difference/m_StDev);
			}
			else
			{
				return 0;
			}
		}
};


ref class TickData
{
	public:
		static Semaphore ^ConsumerSemaphore = gcnew Semaphore(0, 999);
};
```