Algo-trading market client LLD

3 Tier Design

The application will follow the 3-tier design structure:

Presentation Layer

Business Layer

Data Layer

Each layer will have its own namespace or project. A **special PL-BL namespace** will contain shared interfaces and classes that are crucial for both presentation layer and business layer and their communication.

Presentation Layer

**GUI**

**The MainWindow is divided into 4 categories:**  
Market action:  
Includes the buy, sell and cancel queries.  
  
The buy and sell forms looks visually the same and preform each the buy or the sell query. We added a feature allowing the user to calculate the total amount of money he will spend or get when he buy or sell.  
The cancel form allowing the user to cancel a request by the ID of the request.  
  
Info:  
Include the request status, commodity status and user status queries.  
It is also include the history button and a new feature we added "test connection".

**We now enhance the info available to the user by adding Statistics button.**  
The request status form allowing the user to get the a request status by the ID of the request.  
The commodity status form allowing the user to get the a commodity status by number status (the current ask and bid).  
The My status form allowing the user to get his current status (how much he got from each commodity, his funds and his requests ID's). **The user is able now to export his status to PDF. To do that, we used the iTextSharp open source tool.**The history form allowing the user to watch the history of his actions. The user can choose between show by days or dates.   
The test connection button allowing the user to check the most common and also easy to notice problem by himself.

The Statistics form display graphs about each commodity by choice to the user: average price, times reach highest price, times reach lowest price in the last 24 hours.  
  
Default AMA:  
Include the run default AMA and current logics buttons.  
The run default AMA is the button that runs the AMA.   
The Current Logics button reveals to the user the current logics and he can compare it to his own.

**We now added a schedule option to the automatic AMA using open source library call "FluentScheduler". This library include "IJob" class that create a task and then the user can schedule this task to happen whenever he choose. In our case the task is to run the Default AMA.**

User AMA:  
Include run user AMA, add rule, Current logics and clear all rules buttons.  
The run user AMA is the button that runs the AMA.  
The Current Logics button reveals to the user the current logics he set.  
The add rule button allow the user to add a rule by the ask of the commodity.  
The clear all rules button clear the rules the user set.

Business Layer

**HTTP-Clients**

**SimpleHTTPClient Class**

Sends objects using JSON to a server. Receives JSON strings and converts them to objects/strings for the client.

**SimpleCryptoHTTPClient Class**

This class extends SimpleHTTPClient – it adds the nonce to the authentication of the server and decrypts the response.

This class uses the Singleton pattern, as having several existing instances of the class may lead to collisions with the nonces generated.

Nonce Generation

The nonce is a string representing an integer between -9223372036854775806 and 9223372036854775807.

The starting nonce is generated upon creation of the SimpleCryptoHTTPClient. Therefore, using the Singleton pattern is necessary.

In order to make sure that are not using the same nonce, we convert the current date to an integer with 14 digits (up to milliseconds). The 5 final digits are set to zero. Each request sent adds to the counter. This provides us with 99999 requests sent per millisecond. The server allows only 20 requests per 10 seconds - there can be no collisions.

**Server Communications**

**IMarketResponse Interface**

This object represents a response from the market. Any message received, be it a request id or a market exception will implement this interface. The Presentation layer prints out a string from this class to the user.

* **ResponseType getType();**

Return the type of the response.

* **string ToString();**

Returns a string representing the message contained in the response.

**MResponse Class**

Contains several implementations for the **IMarketResponse** interface. Each implementation corresponds with each different response from the server.

**Request Class**

A class representing a request sent by the user. These objects are sent from the Presentation Layer (via GUI) to the Business Layer (via the **InterperatorPB** class)

**RequestType**

This is an Enum for the various types of requests that can be sent to the server.

**ResponseType**

This is an Enum for the various types of responses that can be received from the server.

**InterperatorPB Static Class**

This class receives requests from the Presentation Layer and passes them to the ICommunicator class in the Business layer. It then returns an object signifying the response from the server.

This is the main pipeline for PL-BL communication.

* **public static IMarketResponse sendRequest(Request req)**

Receives a **Request** object and sends it to the server. Returns an **IMarketResponse**

* **public static List<MQCommodityWrapper> sendAllCommodityRequest()**

Sends the new request – Query All Commodities – and returns a **List** of

**MQCommodityWrapper**.

* **public static List<MQReqWrapper> sendAllUserRequest()**

Sends the new request – Query All User Requests – and returns a **List** of **MQReqWrapper**.

**ICommunicator Interface**

This interface handles the communication to the server using a **SimpleHTTPClient**, both input and output.

* **IMarketResponse SendBuyRequest(int price, int commodity, int amount);**

Sends a Buy Request to the server and returns an **IMarketResponse**.

* **IMarketResponse SendSellRequest(int price, int commodity, int amount);**

Sends a Sell Request to the server and returns an **IMarketResponse**.

* **IMarketResponse SendQueryBuySellRequest(int id);**

Sends a Query Buy/Sell Request to the server and returns an **IMarketResponse**.

* **IMarketResponse SendQueryUserRequest();**

Sends a Query User Request to the server and returns an **IMarketResponse**.

* **IMarketResponse SendQueryMarketRequest(int commodity);**

Sends a Query User Request to the server and returns an **IMarketResponse**.

* **IMarketResponse SendCancelBuySellRequest(int id);**

Sends a Query User Request to the server and returns an **IMarketResponse**.

* **List<MQCommodityWrapper> SendQueryAllMarketRequest();**

Sends a Query All Commodities Request to the server and returns a **List** of **MQCommodityWrapper** (one of the implementations of **IMarketResponse**).

* **List<MQReqWrapper> SendQueryAllUserRequest();**

Sends a Query All User Requests to the server and returns a **List** of **MQReqWrapper** (one of the implementations of **IMarketResponse**).

**Communicator Class**

This is the main and most basic implementation of the ICommunicator interface.

It holds 4 fields necessary for the communications with the server:

User id, server url, user private key, http client.

It has 2 different constructors:

* **public Communicator()**

Initializes the fields using default values of user 36.

* **public Communicator(string url, string user, string privateKey)**

Initializes the fields using the received paramaters.

**Communicator Class**

This is the main and most basic implementation of the **ICommunicator** interface.

It holds 4 fields necessary for the communications with the server:

User id, server URL, user private key, simple http client.

It has 2 different constructors:

* **public Communicator()**

Initializes the fields using default values of user 36.

* **public Communicator(string url, string user, string privateKey)**

Initializes the fields using the received parameters.

**CryptoCommunicator Class**

This class extends the **Communicator** class

The difference is that instead of using a **SimpleHTTPClient** for communication with the server, it uses a **SimpleCryptoHTTPClient**.

**TestMarketCommunicator Class**

This class extends the **Communicator** class

The difference is that this class has only 1 empty constructor that sets the URL to be the one of the tests server.

**Autonomous Market Agent**

This class uses the Observer pattern - it holds a queue of **AlgoProcess** and handles the activation of these processes following restrictions from the server (ie: no more than 20 requests every 10 seconds).

The AMA itself holds information from the server – user status, **all** requests status, **all** commodities' asks and bids.

This information is accessed by the each **AlgoProcess** that is activated.

Each **AlgoProcess** is composed by a list of **AlgoConditions** and a single **AlgoAction**, and holds information regarding the whole process – commodity Id, request status…

The **AlgoConditions** access the information stored in the AMA and the **AlgoProcess** to verify if a certain condition is met. If all the conditions are met, then the **AlgoAction** is activated.

**AdvancedAMA class**

This is an upgraded version of the AMA in milestone2:

It holds information from the server – user status, **all** requests status, **all** commodities' asks and bids.

* **public AdvancedAMA(int maxReq, double interval, ICommunicator comm)**

This is the constructor of the class. It receives the maximum number of requests allowed per time interval (in milliseconds). In addition is receives an ICommunicator to use for sending requests.

* **private void OnTimedEvent(Object source, System.Timers.ElapsedEventArgs e)**

Creates a new thread to allow the GUI to run smoothly and does the following:

Call **gatherInfo**()

Subtract the amount of requests sent in **gatherInfo()** from the current request count.

While the current count of requests did not exceed the maximum allowed in an interval

Call **run**()

Add +1 to the current count if an action was active (meaning a request was sent).

* **public bool run()**

Pulls out an AlgoProcess from the queue and runs it's process.

* **public int gatherInfo()**

Sends to the server the following requests:

Query User, Query All Commodities, Query All User Requests. Updates the fields of the AMA accordingly.

* **public void enable(bool toEnable)**

Enable/Disable the AMA.

* **public bool isEnabled()**

Return whether the AMA is enabled or disabled.

* **public virtual void add(AlgoProcess processList)**

Add another AlgoProcess to the AMA queue.

* **public void clearLogic()**

Clear the entire queue.

**Extensions:**

* DefaultAdvancedAMA: Initiates the Advanced with the default parameters allowed by the server (20 requests per 20 seconds) and provides it with a **CryptoCommunicator**.
* DefaultMomentumAMA: Extends **DefaultAdvancedAMA**. Contains an

AlgoCompareBuyProcess + AlgoCompareSellProcess for each commodity.

* DefaultCompareAMA: Extends **DefaultAdvancedAMA**. Contains an AlgoMomentumBuyProcess + AlgoMomentumSellProcess for each commodity.

**Default AMA and User AMA**

The client has a **DefaultMomentumAMA** that acts the client's default AMA.

Using an Open Source project (FluentSchedule), we allow the user to schedule the activation of this AMA.

In addition, the client contains an empty **DefaultCompareAMA** that acts as a User-Configurable AMA. The user is able to add AlgoCompareBuyProcess and AlgoCompareSellProcess to this AMA.

**AlgoProcess class**

* **private void updateRequestStatus()**

Once a request is sent from an AlgoAction of an AlgoProcess, the requests ID is stored in the AlgoPrcess. In this method the AlgoProcess accesses the information in the AMA to see if request is still pending or has been completed. It updates the

* **public virtual bool runProcess()**

Verifies that all conditions are met, and if so – activates the **AlgoCondition** and returns *true*. Else, it returns *false*.

* **public void addCondition(AlgoCondition condition)**

Adds an AlgoCondition to the list of the AlgoProcess.

* **public void setAction(AlgoAction action)**

Sets the AlgoAction of the AlgoProcess.

* **public void clearConditions()**

Deletes all the conditions in the list.

**Extensions:**

* AlgoCompareBuyProcess: Buys a commodity when the Ask is lower than a certain threshold.
* AlgoCompareSellProcess: Sells a commodity when the Bid is higher than a certain threshold.
* AlgoMomentumBuyProcess: Buys a commodity when it is experiencing an **\*upward momentum**. Calculated using **SQL** queries from the server.
* AlgoMomentumSellProcess: Sells a commodity when it is experiencing a **\*downward momentum**. Calculated using **SQL** queries from the server.

**\*upward/downward momentum:**

When a commodities short moving average goes above its long moving average – it is experiencing an upward momentum.

When a commodities long moving average goes above its short moving average – it is experiencing a downward momentum.

**AlgoCondition Interface**

* **bool conditionIsMet(AlgoProcess process);**

verify if a certain condition is met.

**Implementations:**

* AlgoAskCompare: Returns true if the Ask of a commodity is lower than required
* AlgoBidCompare: Returns true if the Bid of a commodity is higher than required
* HasNoActiveRequest: Returns true the AlgoProcess has no pending requests.
* HasSupply: Returns true if the user has a supply of a certain commodity.
* MomentumDecrease: Returns true if the commodity is experiencing a upward momentum.
* MomentumIncrease: Returns true if the commodity is experiencing a downard momentum.

**AlgoAction Interface**

* **bool runAction(AlgoProcess process);**

Sends the appropriate request to the server. Return true if a request has been sent. Return false if otherwise.

**Implementations:**

* AlgoBuy: Sends buy requests using a certain percentage of the user's funds.
* AlgoSell: Returns true if the Bid of a commodity is higher than required

Data Layer

**History**

**HistoryView class**

The history is written into History folder – *Year*.*Month*History.txt.

It uses Log4net to write the history logs into the file using a dedicated appender.

To read the file content the user is given 3 options:

Choose how many backward days, choose a specific range or specific line number.

**Public historyByDate(DateTime minDate, DateTime maxDate)**

Return string array of history log, it return the history logs in the specific given range.

**public historyByLines(int numRows)**

Return string array of history log., it return a specific line number of the most recent history log written .

**public historyBydays(int daysNumber)**

Return history logs string array, it returns history logs by dayNumber days backwards.

**Private getDate(String str)**

Get a string and convert it into a DateTime variable. It is an auxiliary function for the main function in the class.

**public deleteHistory()**

delete the history file content.

**History.bdml**

This is MSLinqToSQLGenerator class that convert the history data base into an item.

All data base connection go'se throw this class

**SQL\_DAL\_implementation**

All functions in this class executes the queries from history data base

In case the query result is empty, in case the return statement spouse to be an item it return null otherwise it return -1

**private itemsByNumDays(DateTime start, DateTime end, int commodity)**

this function execute basic query on the history db by specific dates and commodity.

all the other functions use it.

**public PriceAverage(DateTime start, DateTime end, int commodity)**

this function, return the average price the commodity has bought in the specific date range.

**public PriceAverage(int commodity)**

this function, return the average price the commodity has bought in the last 15 hours

**public highestSell(DateTime start, DateTime end, int commodity)**

this function, return the highest price the commodity has bought in the specific date range.

**public highestSell(int commodity)**

this function, return the highest price the commodity has bought in the last 15 hours.

**public numOfHighest(DateTime start, DateTime end, int commodity)**

this function, return the number of times that the commodity has bought in the highest price in the specific date range.

**public numOfHighest(int commodity)**

this function, return the number of times that the commodity has bought in the highest price in the last 15 hours

**public lowestSell(DateTime start, DateTime end, int commodity)**

this function, return the lowest price the commodity has bought in the specific date range.

**public lowestSell(int commodity)**

this function, return the lowest price the commodity has bought in the last 15 hours.

**public numOfLowest(DateTime start, DateTime end, int commodity)**

this function, return the number of times that the commodity has bought in the lowest price in the specific date range.

**public numOfLowest(int commodity)**

this function, return the number of times that the commodity has bought in the lowest price in the last 15 hours

**public avgPerday(DateTime start, DateTime end, int commodity)**

this function returns an array that contain the average price of the commodity per days in the specific date range.

**public avgPerday(int commodity)**

this function returns an array that contain the average price of the commodity per daysin the last week.

**Log**

In order to write all the app log, we use Log4net.

There is Rolling file appender that create a log text file. There is a separated log file for each month.