1. Introduction

This document is mainly to provide quick explanation of the progress made till now in the AlgoTrader application that is being designed to demonstrate the power of dbToaster. The aim of AlgoTrader was to provide an interface where new algorithms can be tested and simulated in a virtual market scenario, so as to test their efficiency and performance. The development till now has involved designing the market framework, wherein the developer can insert new market components and elements with little effort.

The work till now has resulted in the following components of a Stock Market being completed:

1. The Stock Market itself. The currently implemented market can:
   - Match incoming orders.
   - Provide update feeds. Such an update feed may be sent to selective users only, or to all the people who connect to the stock market. These update feeds may involve order-book updates, match updates, and price updates amongst many others. They also need to notify specific clients when their trades are matched to someone else.
2. A framework for a large algo-trader in the market, who has the right to trade directly with the market (not through some broker or market-maker). The trader can:
   - Parse market updates, and decide if they are order-book updates, or trade updates.
   - Have a mechanism in place which automatically evaluates a few customised parameters on every update, so that an algorithm can access them to make decisions.
   - Decide the algorithm for placing orders in the market.
3. A framework for a market maker. The market maker can:
   - Provide a network detail so that clients can connect to the market maker. At every instant of time, the market maker provides a bid quote and an ask quote to all its clients.
   - Accept orders from its clients, and do the appropriate transaction based on what he offered.
   - Trade in the market, so that it can make a profit from its transactions with its clients.
4. A framework for a broker. This has been left slightly incomplete. The broker currently can:
   - Provide a server connection to its clients. The clients can place orders at the broker.
   - The broker receives feeds from numerous market makers as well as the stock market itself. Based on the feeds, it decides which source has the best price and forwards the clients order to that particular source.
5. A few other minor components include:
   - A collection of threads, to generate a good random stream of orders for simulating history.
   - A modified historic trader implementation which scales historic orders based on how much different the market flow from history is due to addition of an algo-trader.
   - A GUI implementation for plotting a price volume chart of a stock in the market.

First I would like to specify how communication across different market components has been handled, after which I shall describe the implementation of the components specified above.

2. Network Implementation

The entire project has been built using the netty channel APIs. Refer to ⟨hlink | http://docs.jboss.org/netty/3.1/api/ | ⟩ for any explanations. However, I am including the basic idea here so that there is little confusion. For communication using netty, three main things need to be understood clearly:

1. Setting up a server
2. Setting up a client
3. Handling communication

2.1. The Server.

The following is the code to create a general server.

```
ChannelFactory factory = new NioServerSocketChannelFactory(
Executors.newCachedThreadPool(), Executors.newCachedThreadPool());

ServerBootstrap bootstrap = new ServerBootstrap(factory);

bootstrap.setPipelineFactory(new ChannelPipelineFactory());

bootstrap.setOption("child.tcpNoDelay", true);
bootstrap.setOption("child.keepAlive", true);
bootstrap.bind(new InetSocketAddress(8080));
```

For creating any new server, everything remains the same except the third line. The argument to the `setPipelineFactory()` function decides how channel communication (both read and write) will be handled. This is described later.

2.2. The Client.

The following is the code to create a general client:

```
ChannelFactory factory = new NioClientSocketChannelFactory(
Executors.newCachedThreadPool(), Executors.newCachedThreadPool());

ClientBootstrap bootstrap = new ClientBootstrap(factory);

bootstrap.setPipelineFactory(new ChannelPipelineFactory());

bootstrap.setOption("tcpNoDelay", true);
bootstrap.setOption("keepAlive", true);

ChannelFuture cf = bootstrap.connect(
new InetSocketAddress("localhost", 8080));

Channel ch = cf.awaitUninterruptibly().getChannel();
```

Here again, the code for any client remains the except `setPipelineFactory()` function argument which decides channel communication. The last line provides you with a Channel to the server, in case the coder wants to write data to the server through means different than the automatic "factory" implementation passed to the passed as an argument.

2.3. Communication. The `ChannelPipelineFactory` class is abstracted and cannot be instantiated as shown in the above code. To instantiate there is a method that must be implemented. That method is

```
@Override
public ChannelPipeline getPipeline() throws Exception {
return Channels.pipeline(<some handlers>); }
```

This method provides the channel link between the server and client with some handlers for data being sent across between the two. These handlers are all extensions of the class `SimpleChannelHandler` defined in netty. Briefly, this class has overridable functions for the following actions:

1. What do when a data is received from the channel.
2. What to do when a connection is established.
3. What to do when a connection is closed.

And many more. Over-riding these functions to create your Handler, and ChannelPipelineFactory implementation will help you control communication across channels. I will not discuss exactly how to implement the handlers as that has been clearly commented in the examples quoted below.

One implementation of a `ChannelPipelineFactory` is in `stockexchangesim.StockMarketServer.StockMarketChannelFactory` for reference.

One implementation of a handler can be found in `handlers.OrderMatchingHandler`.

3. The Virtual Stock Market

This section discusses the implementation of the stock market simulator, that is the collection of classes that handle the burden of simulating any Stock Market in AlgoTrader.

Let me start of by mentioning the features of the market implementation that must be customisable.

Market Rules. Market rules are the rules which decide price dynamism, compatibility of orders for match, and few other market dynamics (like batched execution or non-batched execution, etc).

Order-Book Structure. By order-books, we refer to the data structure in which the market maintains its pending orders and their various details (such as bid/ask price, volume, trader id, time stamp ,etc). This structure may vary in number and type of fields from market to market.

Parser. This does not vary only from market to market but also from implementation to implementation. It is desirable that the order that is send across a channel to and from the market be customisable. i.e. The string, (or a data structure) that is transmitted across the channel may have different fields, or it may be desirable to change it as the project develops. For this purpose, the parser for such inputs should also be customisable.

It would be very complicated to give a line by line description of the codes in this document. So I shall refrain from specifying minute details and provide the main overview. The details can be picked out by reading the code.

3.1. The Order Matching Framework.

Here I shall describe the framework for the "customisable" market order matcher. The entire bulk of this framework lies inside package `rules` and its subpackage.

Matcher.java . This interface defines the general structure of any market matcher. Basically all the classes of the project that wish to use an order matching object use variables of type `Matcher`. It tries to match an order `a` of type `action`(bids/asks) with any existing entries.

BasicMatcher.java . This is an implementation of the `Matcher` interface, which does NASDAQ market rules based matching of orders. The flow is well commented in the implementation.

UpdateMessageMatcher.java . This is also an implementation of the `Matcher` interface, and behaves just like the BasicMatcher, with the addition that at the end of the matching, it informs all the participant traders of the match of the same. This matcher is used almost exclusively by the stock market alone, since it must produce an update stream for completed orders.

The BasicMatcher on the other hand is used by components of the market who want to simulate a local order-book from market updates, and hence donot need to produce any update messages.

3.2. The Order-Book Structure. This is perhaps the part of the framework that is used by almost every class in the project. The class `OrderBook.java` does the following:

- Provides market related constants. Like which string stands for a bid/ask/delete order, the order format string, etc.
- Defines a class `OrderBookEntry` which contains the relevant fields stored for each order book entry.
- Each instance of this class contains two lists of OrderBookEntry objects for ask orders and bid orders.
- Has overloaded method `createEntry` for creating a new OrderBookEntry for an array of order book entry fields.
- Has a method `executeCommand` for inserting and deleting entries from the bid and ask order books.
- Has accessor methods to obtain the ask or bid order book entries.
- It also has static methods which define the schema of the order book, which is used by other classes like the parser class to decode order strings.

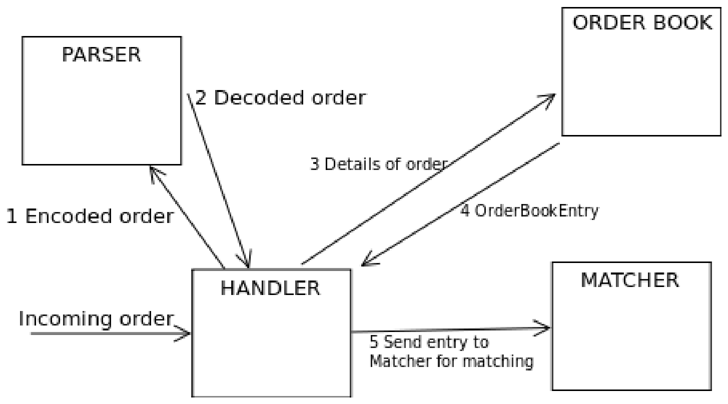
3.3. The Parser. All parsers are defined in the package `codecs`. Currently, we have been using a consistent notation for order strings, hence there is only one parser `TupleDecoder.java`. It takes in a string, and returns a Map<String, Object> where the keys of the map are the keys of the schema defined statically in `OrderBook`.

3.4. The Market. With these utilities in place, the market framework becomes very concise to define.

The `StockMarketServer.java` is an executable, which creates a netty channel server for accepting orders, and creates its own OrderBook, Matcher, and ChannelPipelineFactory structures.

The `OrderMatchingHandler` is the handler used to process new input orders, find their potential matches and create an update stream.

The Matcher object used by the handler is the `UpdateMessageMatcher` which processes every new entry for a match, and sends updates to the respective participants.
The TupleDecoder object and OrderBook object is used to parse and create an OrderBookEntry which the Matcher can use. It is important to note that once the server is created, the execution is actually interrupt triggered, where every network message triggers the netty handlers. The flow is best described by the diagram below



4. The Algo Trader Framework

The framework for an algotrader is very similar to the market in its flow pattern. Since the algo trader being designed functions by processing order-book updates produced by the market, it essentially receives the same trade requests as the market, and in the same sequence.

Firstly, the algo trader must be a netty client connected to the stock market. On receiving a order book update, the market must parse and handle it in the same way as the market. So a trader maintains his own local copy of market Matcher and OrderBook. However, once the order is matched inside the trader, he must do more work. He must evaluate the market scene, and make decisions to trade, and then send a corresponding order sequence to the market.

This additional framework to "evaluate" the market scene is what I shall now describe. We may say that in general, a SOBI trader will keep an eye on the dynamism of certain features with every trade in the market. For example, VWAP(Volume weighted average price) of both bids and asks, their divergence from the current market price and current market trend to name a few. These properties change with every new order.

To quantify these properties and their dynamism, we define a new framework class `General-StockPropts`. This class is abstract, and maintains a local map of String to Object. The String is the name of the property and the Object is its value. For example, to keep track of the market price through this class, we define a map from "price" to Double. From there on, to access the "price" property of the stock, we simply fetch the object for "price" from the map.

So to enable this, the abstract class has methods to add, remove, modify and access these stock "properties". Also the class introduces a notion of general and special properties. General properties are some properties assumed to be needed by every variation of an algorithm. They are more like the basic properties which are needed for every algorithm. Special properties can be added by an abstract method. There is also a method which updates the values of the properties in the map every time the order book changes. The names of the methods and their implementation have been well commented so refer to the code for more details.

The entire algo-trader framework is in package `algotrader.framework`. All implementations of the GeneralStockPropts and GeneralTrader are in the other subpackages of `algotrader`.

5. Market Maker Framework

There is not much to explain about this framework if the design state till now has been clear. The market maker is a slightly more complex framework in terms of network communication as it involves both a netty server and client. It is a client of the stock market server, and acts as a server for clients who want to trade in the market through the market maker.

The market maker does the following:

- Runs a server which displays its bid quote and ask quote from time to time. Any client who wants to trade with the market maker will have to trade at those prices.
- Has a connection to the market, and an algorithm in place for trading with the market from time to time to exploit the market spread.

The `MarketMakerClientChannelHandler` takes care of the orders placed by clients at the MarketMaker, while the `MarketMaker` handles placing trades with the market. `MarketMakerPropts` is the class which contains the details of the market makers functioning, like the market makers quotes, portfolio details, GeneralStockPropts that it uses to make trades, etc. `MarketMak-erServerChannelHandler` is like the usual handler which handles market updates like order-book updates, trade updates, etc.

6. Broker Framework

This framework is incomplete in as much as its network handlers are not completely defined yet. However the remaining framework for the same is defined and commented. Its main purpose is to forward their clients orders to the best target (market or market makers). If the reader has understood the framework so far, then there is nothing new to the codebase for the Broker. Its all about defining the handlers correctly, who's algorithm has been commented in the code.

7. Extras

There are the extra classes that were defined as a part of the testing and running phase. These classes are pretty straightforward. I am just including some extra documentation for the chart drawing GUI.

I have basically use JFreeChart, and tweaked the code to produce a dynamic price cum volume chart which moves with time. The code for creating the chart is in GUI.PriceChart and GUI.PriceVolumeChart. This chart must be created as a thread as it should run as a non-blocking background method. Hence I have created a thread which can be attached to any process(the market, the client, the trader, the market maker, etc) in threads.PriceThread. This thread looks up the StockPrice object associated with its calling process to display a dynamic price and volume flow.