An Introduction To Writing Algorithms *for* Group Alpha Testing Suite

Adam Gleave
Nick Burscough
Artjoms Iskovs
Lawrence Esswood
Christopher Little
Alan Rusnak

1 Getting Started

Version 1.0 of the Testing Suite is almost entirely written in Java. As such, all algorithms that are to be submitted to the system must currently also be written in Java. This example documentation will remain largely independent of IDE, though Eclipse is a good choice due to its ability to streamline JAR creation (as algorithms are submitted to the system as JAR files).
To dive right into algorithm creation, we will now look through the workings of a hello-world-esque algorithm. The algorithm works in a completely random fashion, buying and selling according to the whims of its lord and master, `math.random()`. Obviously, this algorithm will fare extremely poorly in any real-world situation, and serves only to illustrate how we can interface with the testing framework. Following is the algorithm in its entirety, after which we will take a look at a breakdown of how it interfaces with the simulated market.

```java
// HelloRiches.java

import java.util.Iterator;
import orderBooks.Order;
import orderBooks.OrderBook;
import database.StockHandle;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;


public class HelloRiches implements ITradingAlgorithm {

	@Override
	public void run(MarketView marketView, Options options) {
		Iterator<StockHandle> stocks =
		    marketView.getAllStocks().iterator();
		StockHandle bestStockEver = stocks.next();
		OrderBook book =
		    marketView.getOrderBook(bestStockEver);
		double propensityToSell = 4;
		while(!marketView.isFinished()) {
			marketView.tick();
			propensityToSell += Math.random()-0.5;
			if(propensityToSell < 5) {
				//BUY BUY BUY
				Iterator<? extends Order> iter =
				    book.getAllOffers();
				if(!iter.hasNext()) continue;
				int bd = iter.next().getPrice();
				marketView.buy(bestStockEver, bd, 1);
			} else {
				//SELL SELL SELL
				Iterator<? extends Order> iter =
				    book.getAllBids();
				if(!iter.hasNext()) continue;
				int bd = iter.next().getPrice();
				marketView.sell(bestStockEver, bd, 1);
			}

		}
	}
}
```

2 Breakdown of HelloRiches.java

2.1 Imports

```
import orderBooks.Order;
import orderBooks.OrderBook;
```

The orderbook construct holds all bids and offers for a stock. Each stock has an associated order book. An order is simply one of these bids or offers. We will see later how we can use these classes to place our algorithm's offers onto the market and also bid for existing orders.

```
import database.StockHandle;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;
```

A stock handle essentially does what it says on the tin and allows us to hold a reference to a particular stock so that we interact with it (buying or selling etc). Market View and Options exist to allow the testing framework to test market situations and vary simulation options such as tick size. Finally, the ITrandingAlgorithm class is implemented by the algorithm to ensure correct interfacing with the framework (i.e. the presence of a `run()` method.

```
@Override
public void run(MarketView marketView, Options options) {
```

As we are implementing the `ITradingAlgorithm` interface, we must `@Override` the `run()` method. The `MarketView` and `Options` will be decided by the testing server and handed to the algorithm.

```
Iterator<StockHandle> stocks = marketView.getAllStocks().iterator();
StockHandle bestStockEver = stocks.next();
OrderBook book = marketView.getOrderBook(bestStockEver);
```

First, we define a new `Iterator` that iterates over all possible stocks - (`marketView.getAllStocks().iterator()`), and then use it to arbitrarily pick a stock for our algorithm to work with. In this example we simply choose the stock that is returned from `stocks.next()`, i.e. the first stock

of the iterator. Finally, we extract the `OrderBook` for this specific stock so that we can begin attempting to interface with the market.

```
double propensityToSell = 4;
while(!marketView.isFinished()) {
        marketView.tick();
        propensityToSell += Math.random()-0.5;
```

Here we begin to set up the logic of our algorithm - how we are going to determine our money making strategy for this market tick. A tick, in this context, refers to the smallest possible snapshot of time that our algorithm can witness in our market. We can do as much computation as we need to (provided we do not time-out our simulation) within a tick and then advance in time by calling `marketView.tick()`. For this example algorithm, our strategy is going to be based on the rolling of `Math.random()` affecting our inclination towards selling the stock we have bought. In this code snippet specifically, we see that, so long as the simulation has not finished, we will advance to the next tick of time and then alter our inclination towards buying/selling with equal weight.

```
if(propensityToSell < 5) {
        //BUY BUY BUY
        Iterator<? extends Order> iter = book.getAllOffers();
        if(!iter.hasNext()) continue;
        int bd = iter.next().getPrice();
        marketView.buy(bestStockEver, bd, 1);
} else {
        //SELL SELL SELL
        Iterator<? extends Order> iter = book.getAllBids();
        if(!iter.hasNext()) continue;
        int bd = iter.next().getPrice();
        marketView.sell(bestStockEver, bd, 1);
}
```

Directly following from the previous snippet, after altering our propensity to sell, we then check if we lie within the buying range or the selling range. Note that the initial value of `propensityToSell` is set to 4 to force the market strategy of the first tick to be a buy (as we of course need to buy before we can sell). Upon making our buy/sell decision, we use the `OrderBook` object `book` to get the iterator for either all current bids or all current offers

for our chosen stock. Then, we check to see if the iterator is non-empty, setting an int to the value of either the best off or the best bid if this is the case. Finally, we use our `marketView` to buy or sell respectively.

(And then progress to the next `tick()`, as per our loop)

3 Key Commands/Concepts

The simple example covered a few of the methods needed to interface with the market, but here we will concretely list everything you need to spread the wings of your algorithm and fly slightly farther afield.

4 A (more in depth) example

Yet to be written